Universidad Autónoma de Madrid
Escuela Politécnica Superior

Thesis to obtain PhD. degree in
Computer Science and
Telecommunication Engineering
by Universidad Autónoma de Madrid

Thesis advisor: Dr. Eloy Anguiano Rey

Platform for automatic paralellisation of sequential codes using dynamic graphs partitioning and based on user-adaptable load balancing

Carmen Blanca Navarrete Navarrete

This thesis was presented on 2011
Tribunal:

Dr. Michael Gerndt
Dr. José María Carazo
Dr. Jesús María González Barahona
Dr. Iván González Martínez
Dr. Alfonso Ortega de la Puente

Carmen Blanca Navarrete Navarrete
Platform for automatic paralellisation of sequential codes using dynamic graphs partitioning and based on user-adaptable load balancing

Carmen Blanca Navarrete Navarrete
Escuela Politécnica Superior. Dpto Ingeniería Informática

IMPRESO EN ESPAÑA – PRINTED IN SPAIN

Thesis evaluators:

Dr. Michael Gerndt
(Chairman)

Dr. José María Carazo

Dr. Jesús María González Barahona

Dr. Iván González Martínez

Dr. Alfonso Ortega de la Puente

Reviewer:

Dr. Francisco Javier Gómez Arribas

Index

ACKNOWLEGMENTS

Sometime ago, somebody told my the saying "well born is to be grateful" so, lets start with the acknowledgements.

First of all, I will like to thank my two servers *memnon* and *fobo* for these hard five years of configurations, compilations, tests, core dumps, reboots... without any reproaches. They are the seed of all the work done here. From the union of servers and efforts, other computers were born, as *maya*, the first cluster controller or *metis*, the first multi-core machine of the lab. But really, the thanks must be given to *rayhalle2* and *hlrb-II* nodes, the ones which have suffered the last two years of experiments. And as I am of the opinion that in average once each 100 pages a joke must be done, here were my contribution and a half more.

Agradezco también este trabajo a las instituciones UAM, IBM e IIC mediante la financiación de la asistencia a congresos y en general de los recursos usados durante este período.

Ich würde auch Inés (“Kindergarten”, wie sie immer gesagt hat) und Sabine (“Versuchen Sie es nochmal”) dafür bedanken, dass Sie mir in der deutsche Sprache eingeführt haben. Neither I can forget the support given by Esmé, who made me remember how to trick in English language.

Without further delay, I will like to thank to all my colleagues of CRL, specially, but not only, to Iñaqui who has somehow followed my steps but improving them, to Javi for the suggestions while coding, to Fernando (Fer) who shares this little hobby with me, to Paul (my mirror somehow!), Pablo, Germán, Luis (Luxo), Teso, Alfonso and Yolanda. Von CRL will ich auch in besonders Stephan bedanken, dass er genüg Gedulg hat, um jeden Tag zu versuchen, meine Deutsch zu verstehen und zu verbessern.

And what about friends from different walks of life, far and near! In no particular order, Antonio, Nieves, Mae and Víctor (lets play!); Antonio Larrosa (the Qt version is on the road!); Augusto, Juanjo (Lady Halcón) and Maite; Joseba, to whom I owe my C++ knowledges, and Araceli; Julio (words of the day) and Arkaitz; Laura and Carlos (the “Petits Suisses”); Marcel und Anca, für Ihre freundlichkeit, als ich im Müchen war; Tania and Guille (that are not coloured squares moving around!); Ricardo (you made it!), Manu (you either!), Fernando Herrero, Carlos Castañeda (it’s your turn!), Atienza, Bruno and many other I am probably forgetting.

Ich bedanke auch denen, die meine beiden Forschungsaufenthalte in Deutschland möglich

gemacht haben: Herr Prof. Ulrich Rüde und die Leute des Lehrstuhl für Simulation der Erlangen-Nürnberg Universität; und die Leute des Instituts für Informatik I10 der Technische Universität München, in besonders Herr Prof. Michael Gerndt für Ihren Raten und Unterstützung.

Desde el punto de vista académico, tengo también que agradecer a mi tutor Eloy Anguiano los comentarios aportados, en especial las correcciones y revisiones realizadas hasta llegar a esta versión definitiva además del estilo de LaTeXde este documento; y por otro lado, a Alfonso Ortega, por haber probado la plataforma y las diversas sugerencias realizadas.

Por último, aunque ni mucho menos con menor énfasis, agradecer y dedicar esta tesis a mi familia, en especial a mis padres, María Elia y José María, que son los que realmente me han visto día tras día “sentada tecleando frente a una pantalla negra” e intentar entender todo el escrito que viene a continuación; a mis hermanos, Esther y Jose, por todos los momentos compartidos de apoyo, inspiración y cachondeo, sin los cuales esto hubiese sido muuuucho más largo; a Juan, por todas las horas echadas en viajes, congresos, estancias, charlas, ensayos... tomando notas de los fallos; a Alberto y Norma, por la parte que les corresponde de mis hermanos; y a mi pequeña “sobri”, Candela, que aunque aún no entiende nada de esto, en algún momento se lo leerá, (padres, ¡obligadla!).

Y a los que, sin haber nombrado para que esto no se haga infinito, han aportado ideas, sugerencias, críticas... a este trabajo.

Abstract

In recent decades, both the computational power of processors and the physical transfer rates have grown very fast, allowing the large advance and the improvement of parallel computing. Increasing the computing power and the transfer rates to continue running the same algorithms, that were created to be executed sequentially and on a single processor machine, is completely meaningless [1] for the parallel computing science. It is therefore necessary to rewrite these algorithms, so greater advantage of these new technologies can be taken. Rewriting these applications to their equivalent concurrent version and optimize them implies depth knowledge both about the code of the application, the synchronisation policies between processes, checkpoints, shared data objects as well, and about an extensive knowledge on networks architecture and hardware systems. It is also needed to consider that depending on the solution taken to decompose the problem into at least as many domains as processes, a better or worse performance will be obtained and therefore, a better use of the resources of the cluster; choosing an inappropriate domain decomposition will drastically affect the speed up of the parallel solution.

The solution for avoiding these problems, without developing specific parallel applications for each algorithm that runs in the cluster, goes through the use of a generic platform able to execute existing sequential codes in a parallel way, discovering dynamically on runtime the parameters that optimize the application performance in the cluster from the point of view of the resources of the system, as the network and data topology of the nodes that are part of the cluster.

This documentation is organised in several parts, from the most theoretical one until reaching the tests that have been performed and the results achieved. At the beginning of this document, an introduction to the proposed solution is presented, where the most important theoretical and mathematical concepts can be found, as well as the motivation and features of the developed framework. In a second chapter, the state of the art of the parallel computing is introduced, regarding also the state of the art of the load balancing techniques, the methodology for the domain decomposition and the state of the actual techniques for the automatic parallelisation. These two chapter of the document form the “theory”, whereas for the “practice” are the next parts reserved.

The practical part is afterwards included, beginning with the most explanatory chapter, the methodology. This part covers from the architecture of the platform until the explanation on how the system works, leading into the last part of the thesis, where specific problems have been studied. These problems are presented as test applications, proof of concepts of

the features and power of the platform developed for this work.

In the second volume of this work the reader will find the appendices with more advanced technical information and specifications at a low level of the APIs implemented and used.

Resumen

Durante las últimas décadas, el incremento de tanto la potencia de cálculo de los procesadores como de las tasas de transferencia por medios físicos, ha dado lugar a un gran avance y mejoras en el mundo de la computación paralela. Incrementar la potencia de cálculo y las tasas de transferencia para seguir ejecutando los mismos algoritmos que fueron creados para ser ejecutados de forma secuencial y en un único procesador, carece completamente de sentido [1]. Es por ello necesario reescribir aquellos algoritmos, de forma que se pueda obtener un mayor beneficio de estas nuevas tecnologías pero, reescribir estas aplicaciones para mirarlas a su equivalente concurrente y su correspondiente optimización implica un amplio conocimiento no solo acerca del código de la aplicación sino también sobre las políticas de sincronización entre procesos, puntos de control, objetos distribuidos... y un conocimiento también amplio acerca sobre arquitectura de redes y sistemas hardware. Es también necesario considerar que, dependiendo de las soluciones que se tomen para la descomposición del dominio del problema dado en al menos tantas particiones como procesos esclavos contribuyan a la resolución del problema, se obtendrá un rendimiento mejor o peor de la aplicación en el sistema distribuido y por tanto, un mejor uso de los recursos disponible en el cluster; elegir un método de particionamiento del dominio puede afectar drásticamente al rendimiento de la solución paralelizada.

La solución para evitar estos problemas, sin tener que desarrollar aplicaciones específicas para cada uno de los algoritmos que se ejecutan en el cluster, pasa por el uso de plataformas genéricas capaces de ejecutar códigos secuenciales ya existentes de forma paralela que descubran dinámicamente y en tiempo de ejecución tanto los parámetros propios que optimizan el rendimiento de las aplicaciones desde el punto de vista de los recursos del sistema como la arquitectura y topología de red (ver topology) de los nodos que forman el cluster.

La documentación de este trabajo se organiza según distintas partes, desde la parte más teórica hasta la práctica en la cual se presentan las pruebas realizadas y los resultados obtenidos. Al inicio del documento, se presenta la introducción a la plataforma propuesta, donde se detallan los conceptos más importantes desde el punto de vista tanto teórico como práctico, así como la motivación y capacidades de la plataforma desarrollada. En un segundo capítulo de esta documentación se introduce un estado del arte de la computación paralela, con respecto también a las técnicas de balanceo de carga, las metodologías existentes sobre la descomposición de dominios y el estado actual de las técnicas de paralelización automática. Estos dos apartados del documento conforman la teoría mientras

que para la práctica se reservan los siguientes apartados.

La parte práctica se incluye a continuación, comenzando con el capítulo probablemente más aclaratorio, la metodología. Este apartado abarca desde la arquitectura de la plataforma hasta la explicación de cómo funciona el sistema globalmente, dando lugar así a la última parte de la tesis donde se estudian diversos problemas específicos. Estos problemas han sido expuestos como aplicaciones de prueba, prueba de conceptos de las funcionalidades y potencia de la plataforma desarrollada para este trabajo.

En el segundo volumen de este trabajo el lector podrá encontrar los apéndices, con información técnica más avanzada y las especificaciones a bajo nivel de los distintos APIs implementados y utilizados.

1 Introduction

The aim of this work is to present a platform that has been developed to allow users to run sequential source codes in a parallel way, regardless of the matter of the algorithm and the topology and architecture of a certain cluster of computers (group of linked computers, working together closely thus in many respects forming a single computer).

The first section of this introduction is concerned with the motivation for the present work. The second section lists the theoretical issues that are encountered when analysing the problem. A short description of the mathematical approach is then presented. Finally, the features and main characteristics of the proposed framework provides the reader with a bird's view of on what consists the platform.

1.1 Motivation

It was September 2005 when I presented my master thesis on Computer Science. This work was related to parallelisation of methods to simulate the growing of surfaces of certain materials based on a Montecarlo method simulation. I started from an old sequential code, result of a complex model from the standpoint of simulation, given the number of parameters involved in the growth of a surface, and the high processing power needs, translating this into high run-times and a poor performance.

The algorithm used to perform the simulation of the growth of a layer from the viewpoint of the theoretical behaviour consists basically of four steps, repeated consecutively until the required value to cover the equivalent of a mono-layer is reached:

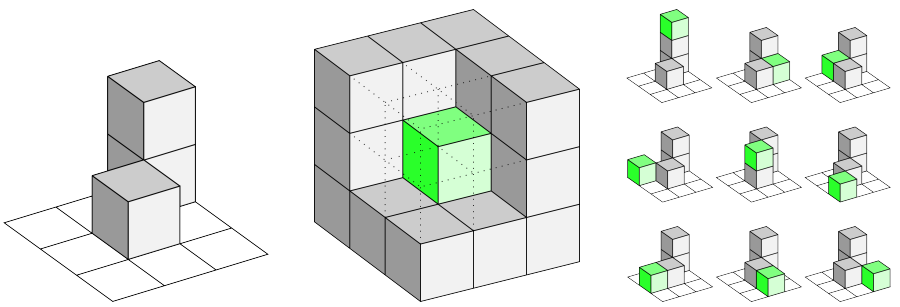
1.– Deposition of an atom in a random position in the surface.

2.– Calculation of the elementary probability of diffusion. If the diffusion takes place, the simulation continues with the step 3. Otherwise, the atom is placed in that position and the algorithm returns to step 1.

3.– Calculation of the energies of the first neighbours. According to the value obtained, the direction of the jump is calculated.

4.– Movement the atom to its new position and return to step 2.

When an atom diffusion simulation is taking place, the atom can leave the simulation surface limits. As the probability of an atom leaving the simulation surface is the same than probability of another atom entering in the simulation surface, it is possible to assume BvK (Born-von Karman) type boundary conditions, and therefore, when an atom leaves a simulation area limit, enters by the opposite limit. Once an atom is deposited in the surface original position (see figure 1.1(a)), the elemental diffusion is calculated. For this, the energy that the atom would have for each possible diffusion position from a known one must be computed (see figure 1.1(c)). With this energies, the partition function that determines the probabilities of transition is calculated and according to partition function, the elemental diffusion that will take part in this atom is randomly determine. It is necessary to keep in mind that the calculation of the energy at at one specified cell of the lattice, the neighbours located at a cube of dimensions $3 \times 3 \times 3$ centred on the actual position are involved (see figure 1.1(b)).



(a) An initial distribution of atoms previous to the deposition of an atom in the centre position of the grid.

(b) First neighbourhood area that contributes energy to the central atom.

(c) Accessible positions of an atom which diffusion phase is taking place. The final position would depend on the bounding energy of each marked position.

Figure 1.1: Procedure to deposit an atom in a surface and diffusion over it.

Even then at that time, I realised there was a lack on the capabilities of the application regarding the domain decomposition and load balancing features. The problem was parallelised following an standard master–slave model, where the master process played the role of conductor of the orchestra whereas the slave processes were in charge of executing the diffusion method along the portion of the surface received. The diffusion problem was simulated using several domain decomposition methods, adding the difficulty that the flow

of depositions and the length of the jump in the diffusion phase is controlled by the global temperature of the environment and the substrate, as it is determined by the thermodynamic equations. In this way, the domain decomposition method depends on a variable that may not remain constant, entailing bigger or smaller environments and thus, different partitions depending on the conditions of the problem. The first question was then presented, why should the domain decomposition remain constant if the physical parameters of the problem were changing? But, there were even a more complex problem to solve: if the flow of depositions were continuous and the positions of the impacts were randomly chosen, why should a slave wait until the deposition carries out in another portion of the surface, breaking the continuity of the flow of depositions? This problem was solved by the implementation of queues connecting the master process with each of the slaves, each one sized to a value related to a load balancing parameter obtained as a function of the capabilities of the node. This solution allowed different growth rates between the different portions of the surface, thing that takes place in Nature.

And, what happens if the network topology does not allow a process to directly communicate with another one that has the values needed to complete its computations (data dependencies)? and if the data dependencies changes while the simulation is taking place? A clustering of the information needed by each slave must be done, information that will be represented as a graph and mapped as the network topology. What happens if a node falls down?, who takes care of its actions and its results?

All those questions were answered implementing multiple solutions, turning the application more and more complex, being then difficult to compare the parallelised algorithms and the sequential and original version.

After this application, I started to apply the same concepts of the diffusion algorithm to a very similar problem, the Potts problem (refer to section 6.3) but, although the model was similar, the symmetries of the problem were different. Why should a programmer develop another complete parallel solution when the models are almost the same and the network topology and load balancing policies were identical?.

These questions prompted us the fact to develop a generic platform that allows the execution of existing source codes in a parallel way, which allows the dynamic reconfiguration of the application based on external factors not related to the application as itself, but to cluster of computers.

1.2 Theoretical Issues

The main problem designing a parallel or distributed algorithm lies in the communication and synchronisation of all processes, needed for the concurrent execution on different processors. From the point of view of synchronisation, the programmer of parallel and distributed applications has certain communication mechanisms between processes, since the introduction of the BSD socket interface in operating systems derived from the original UNIX kernel, such as for example, UNIX System V, Sun Solaris, IBM AIX and HP-UX. Socket primitives allow communication between processes on UDP (User Datagram Protocol) and TCP (Transmission Control Protocol) protocols, providing the software to solve the basic problems of parallel and distributed programming. However, programming with sockets offers a low-level knowledge and forcing the programmer to know the characteristics of the communication network, definition of data structures, synchronisation between communicating processes, etc. This type of low-level programming systems may be suitable for small or non-complex problems, but its features, complexities and limitations make it unsuitable for the development of large parallel applications [2]. These methods add more difficulties making the system scalable.

Libraries based on message passing provide a higher level of abstraction to implement the communication and synchronisation between parallel processes, solving the problems of handling data types, fault tolerance, security mechanisms, among others. In these libraries, multiple routines for data transfer and synchronisation have been integrated and thanks to them, the programmer can ignore the complexities inherent of low-level mechanisms (sockets), transport protocols, etc). There are many examples of such libraries as: PVM (Parallel Virtual Machine) [3], MPI (Message Passing Interface) [4], PVMPI [5], HeteroMPI (Heterogeneous MPI) [6], OpenMP [7], etc.

Parallel processing techniques are then, a set of methods designed to solve problems on systems with a higher processing power, higher than the traditional model of von Neumann computer. These systems, known as multiprocessors, allow to solve too complex problems, impossible to be solved by the traditional paradigm.

The performance of a parallel application is strongly bonded to the concept of parallel overhead. It is mainly related to the communication and synchronisation of processes.

The main feature of multiprocessor systems is to have a set of interconnected processing units through a physical environment that enables data communication and control between them. These communications are not always needed and depends upon the problem to solve. There are a number of important factors to consider when designing a parallel program on behalf of the inter-task communications. These factors can be summed up on the following ones:

1.– Costs of communications

- Inter-task communication always implies overhead. There is no way to communicate two instances (nodes, machines...) without paying an overhead.
- Machine cycles and resources that could be used for computation are instead used to package (serialise and deserialise the information) and transmit data.
- Communications frequently require some type of synchronisation between tasks, which can result in tasks spending time on waiting states instead of doing work.
- Competing communication traffic can saturate the available network bandwidth, further aggravating performance problems.

2.– Concepts of latency and bandwidth

- Sending many small messages can cause latency to dominate communication overheads. It is generally more efficient to package small messages into a larger message, thus increasing the effective communications bandwidth.

3.– Traceability of communications

- With the message passing model (see page 20), communications are explicit and generally quite visible and under the control of the programmer.
- With the data-parallel model (see page 21), communications often occur transparently to the programmer, particularly on distributed memory architectures. The programmer may not even be able to know exactly how inter-task communications are being accomplished.

4.– Kind of synchronism

- Synchronous communications require some type of "handshaking" between tasks that are sharing data. This handshake may happen in code at a lower level explicitly done by the programmer or even in a lower level unknown to the programmer (parallel library).
- Synchronous communications are often referred to as blocking communications since other work must wait until the communications have completed.
- Asynchronous communications allow tasks to transfer data independently from another one.
- Asynchronous communications are often referred to as non-blocking communications since other work can be done while the communications are taking place.

5.– Scope of communications. Knowing which tasks must communicate with each other is critical during the design stage of a parallel code.

- Point-to-point communications involves two tasks with one task acting as the sender/producer of data, and the other acting as the receiver/consumer.
- Collective - involves data sharing between more than two tasks, which are often specified as being members in a common group, or collective.

6.– Efficiency of communications. Depends completely on the application.

- Which implementation for a given model should be used? The MPI implementation may present a better performance on a given hardware platform than on another.
- What type of communication operations should be used? Asynchronous communication operations can improve overall program performance but are not the “golden solution”.

The most noteworthy idea of the parallel computing is to decompose the problem into sub-problems that are easier to solve; that is, the *Divide and Conquer* philosophy . But, it is needed to consider that depending on the solution taken to decompose the problem into at least as many domains as processes, we will obtain a better or worse performance and therefore, a better use of the resources of the cluster; choosing an inappropriate domain decomposition will affect the speed-up of the parallel solution but, the domain decomposition depends both on the problem that we want to execute in the cluster as well as on the symmetries of it. Thus, the idea is to develop a generic platform for the execution of existing sequential codes, so that the parameters that optimize the application performance in the cluster (as network and data topologies or domain decomposition...) will be dynamically obtained and simultaneously with the execution of the algorithm.

In general, the domains decomposition algorithm must take into account the properties and symmetries of the problem and must change them if the speed-up decreases.

1.3 Mathematical Model

The use of clustering computing to solve computational problems has been the focus of the high performance computing community for more than two decades. For general optimisation of hosts in farms of computers, a methodology of dynamic resources allocation is needed [8].

It is important to mention here some mathematical concepts that are going to be used in this work at length, in order to presents the basic graph theory needed to understand the remainder of this work; an interested reader is pointed to [9] or [10]. It is about graph theory: graph, edge and vertices.

Graphs of different types are all around us. Road maps can be seen as collections of towns and roads that link them. Computer networks can be analysed as boxes and cables. Hierarchies, flow diagrams and biological interactions can also be represented as sets of items, vertices, connected to each other by edges. Although the use of graph-like representations is much older, the formal characterization of graphs as abstract mathematical entities can be dated back to the 18^{th} Century and with the famous problem that the mathematician Leonhard Euler set out, the "Seven Bridges of Königsberg" problem (see figure 1.2). Since then, graphs have found a widespread use in many branches of mathematics, most importantly the field of "Graph Theory". Graph theory is a sub-field of Discrete mathematics. It is related to the fields of Combinatorics, Group Theory, and Topology.

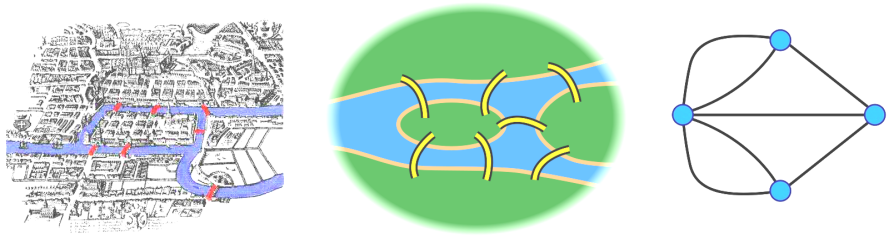


Figure 1.2: The city of Königsberg in Prussia was set on both sides of the Pregel River, and included two large islands which were connected to each other and the mainland by seven bridges. The problem was to find a walk through the city that would cross each bridge once and only once. The islands could not be reached by any route other than the bridges, and every bridge must have been crossed completely every time; one could not walk halfway onto the bridge and then turn around and later cross the other half from the other side. Euler proved that the problem has no solution.

From a mathematical standpoint, a graph G is a collection of vertices V and edges L, where each edge $l = (u, v)$ connects two vertices $u, v \in V$: $G = \{V, L\}$ with $L \subseteq V \times V$. In $l = (u, v)$, u is the source vertex of the edge, and v is the target one. This can be written as $source(l) = u$ and $target(l) = v$, respectively. Some authors prefer to use links instead of

edges, but in the case of this work, the concept *link* will be reserved to the communication of two nodes of the cluster.

If the order of vertices on an edge is not considered important (and L is considered a set of unordered pairs $\{u, v\}$), the graph is said to be undirected. Undirected graphs are equivalent to directed graphs where, for each undirected edge $\{u, v\}$, edges (u, v) and (v, u) can be found. An undirected version of a directed graph can be built by substituting, for all pairs of vertices u and v with at least one edge between them, any directed edges (u, v) or (v, u) by a single undirected edge, $\{u, v\}$.

If $G = (V, L)$ is a graph, a graph $G_1 = (V_1, L_1)$ is a sub-graph of G if V_1 is non-empty and $V_1 \subseteq V$. Since G_1 is also a graph, it follows that L_1 can only contain edges that were present in L, and only those whose endpoints are still present in V_1. It may, however, include less edges than it could. If L_1 includes all edges present in G for the chosen subset of vertices V_1, then G_1 is said to be the an induced sub-graph of G. A special notation exists for this case. If U is a subset of the vertices in V, then $\langle U \rangle$ represents the sub-graph of G induced by U.

If x and y are two vertices (not necessarily distinct) in V, a x, y walk is a finite sequence of vertices and edges such that each edge in the sequence takes off where the last one left. When a walk has no duplicate intermediate vertices, it is said to be a path. The set of vertices that are connected through a path of length 1 to a given vertex form the neighbourhood.

Hence, the cluster or network of computer systems can be modelled as a weighted and not directed graph G_s, denoted by:

$$\boxed{G_s(P, L, \tau, \delta)} \tag{1.1}$$

referred as the *System Graph*; P denotes a finite set of processors that represents the nodes or vertices of the graph G_s; L is a finite set of links that represents the communication links between pair of processors: the edges of the graph G_s; Each vertex $p_i \in P$ is characterised by a set of system parameters (memory, frequency...), based on its available resources of the cluster. Due to this, each processor has a processing weight $\tau(p_i)$ that denotes the processing cost per unit of computation. The nodes allocate information related to the effective load of the processor, defined as its availability to execute other processes. The nodes also contain a statistic value proportional to the execution time of the processes in the previous steps of the algorithm.

Each link between two processors p_i and p_j, denoted by $l_{i,j} \in L$, has a link weight $\delta_{i,j}$ that means the communication latency between those two nodes per transfer unit. If two nodes $(p_i, p_j) \in P$ are not connected to each other, then $l_{i,j} = \infty$. It is assumed that all nodes of the graph are connected to at least one node of the NoC (Network of Computers) (connected graph) but any constraints on the network topology are enforced, as this is not

completely defined and they can vary between two steps of the simulation. It is necessary to define a neighbourhood function which will return the set of nodes that are linked with any node of the cluster:

$$\forall p_i \in P : neigh(p_i) = \{p_k\} \mid l_{i,k} \neq \infty \tag{1.2}$$

Therefore, the edges of the graph represent the connections between the available processors in a certain instant of the algorithm. These connections are statistically weighted as a function of the communication overhead: the time of transmission of the data through the net (which also depends on the network protocol), and the physical layer used for this communication. These two values are important for the calculation of the heuristic function.

The application can also be modelled as a weighted and directed graph G_a, denoted by:

$$G_a(T, D, \omega, \lambda) \tag{1.3}$$

referred as the *Application Graph*, where T denotes a set of vertices of the graph that represents the tasks to be done and D represents a finite set of edges of the graph where $\{(t_i, t_j) \mid t_i, t_j \in T\}$. Each vertex has a computation weight $\omega(t_i), \forall t_i \in T$ that represents the amount of computations required by the task t_i to accomplish one step of the algorithm. Each edge has a value $\lambda_{i,j}$ that represent de amount of data to be sent from v_i to v_j. The existence of an edge between vertex A and vertex B, means that, to calculate the value of A at a certain instant of the execution, we need the value of B at the previous step of the algorithm. We say that A has a data dependency with B.

Thus, the execution time Γ of a task $t_i \in T$ on a processor $p_j \in P$, assuming the worst case in which there is no-overlapping between computation and communication, is defined as:

$$\Gamma(t_i, p_j) = \omega(t_i) \times \tau(p_j) + \sum_{t_n \in neig(p_k)} \sum_{\substack{p_k \in P \\ k \neq j}} \lambda(t_i, t_n) \times \delta(p_j, p_k) \tag{1.4}$$

where $\omega(t_i) \times \tau(p_j)$ represents the amount of computation required by the task t_i per processing cost per unit of computation at processor p_j, the set t_n contains the neighbours of the vertex p_j of the graph G_s, P is the set of vertices of graph G_s, $\lambda(t_i, t_n)$, represents the amount of data to be sent from t_i to t_n and $\delta(p_j, p_k)$ is the weight of the link between processors p_j and p_k.

In order to build a computing environment, it is needed to have an algorithm that has the ability to predict the performance and the resource consumption, considering different cluster configurations. This algorithm is called the RMS (Resources Management System) and it is in charge of the application workbalance. The problem is that it is difficult to predict the computing time per processor, before it receives some arbitrary load, and even more difficult if we

consider the variation of the communication topology of the NoC. The main task of the RMS algorithm is to adapt the amount of work τ for each process depending on some characteristics. The RMS uses a predictive estimation based on a mathematical function (heuristic function, Υ) in order to map the tasks of the parallel application to the pool of resources. The techniques for predicting the performance of the dynamic system are nowadays based on queueing techniques and/or on historical techniques. Making a comparison between layered queues and the historical model, it is well known, that the layered queue requires more CPU (Central Processing Unit) time to make the mean response time prediction, whereas the historical predictions model are almost instantaneous. However, queuing techniques are easier to implement, than the historical model, with a minimum level of performance. This is because designing an historical model involves specifying and validating how predictions will be made, whereas the queueing model can be solved automatically.

Given a system graph G_s and an application graph G_a, the objective is to map their characteristics $\Gamma : (T, D) \mapsto (P, L)$ to minimize function Γ, based on the application requirements and the system constraints, such as the topology of the system graph.

1.4 Platform Overview

The *framework* as a concept is an abstraction in which software providing generic functionality can be selectively changed by user codes (plugins), thus providing application specific software. It is a collection of software libraries providing a defined API (Application Program Interface). This concept is very similar to the one of library but, unlike libraries, in frameworks:

- The overall program flow of control is not dictated by the caller, but by the framework as it self.
- Can be extended by the user, usually by selective overriding of methods or specialized by user code providing specific functionalities.
- The code, in general, is not allowed to be modified. Users can extend the framework, but they can not modify its code.

Software frameworks consist of *frozen spots and hot spots* [11]. *Frozen spots* define the overall architecture of a software system that is, its basic components and the relationships between them. These remain unchanged in any instantiation of the application framework. *Hot spots* represent those parts where the programmers using the framework add their own code to extend the functionality specific to their own project.

The general approach is to part the problem into independent subtasks, in order to minimize communication and synchronisation among processors. Independence among tasks would also allow the introduction of fault tolerance mechanisms more easily than in distributed applications with a complex communication pattern. A partition strategy has been adopted and discarded those solution based on a collaborative approach among processes, such as distribution techniques similar to the ones proposed in [12] and [13], more suitable for dedicated HPC (High Performance Computing) systems.

The approach we propose is based on three main aspects:

- A search space partitioning strategy.
- Distributed task with dynamic load balancing.
- A peer-to-peer communication framework.

A static load balancing policy cannot be adopted as the work load is not known in advance and cannot be estimated. Each slave process has to register itself at a structure allocated at the master process (system graph G_s) to join the system and to contribute to the overall computation task. Peers cannot directly exchange system information and computation subtasks, since each process can only communicate with its parent process.

1.4.1 Features and Limitations of the platform

The goal of the platform is the way it solves the possible collisions between values over frontiers, situation in which one node, to update the current grid position, needs some information that is placed in another node of the cluster.

The proposed framework presents several features as well as limits but, despite of its limitations the platform is powerful enough to run complex user algorithms, as it has been demonstrate in chapter 6.

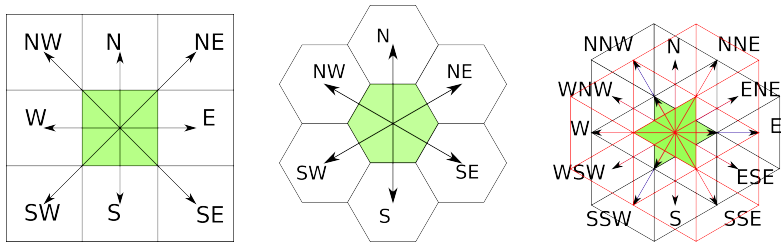
Features

The platform presents the following features:

- API implemented in ANSI (American National Standards Institute) C++ language. It uses the MPI-2 extension.
- Designed as a standard master-slave approach.
- Allows the execution of sequential code in homogeneous(the one which all of its nodes have the same architecture, operating system, hardware features and key components libraries) or heterogeneous(the one that is not homogeneous) clusters, taking the advantages of the workload and dynamic partitioning concepts, with no need to rewrite the sequential code to adapt it to the new parallel environment.
- Adaptable to the three different tessellation that fill a 2D space with no overlaps or gaps (see figure 1.3).
- Master side transparent for the user.
- Plugins for the work balance policies and for the domain decomposition method.
- Support for user defined plugins, apart from the system default plugin sets.
- User defined datatypes.
- Configurable platform behaviour using user .conf files.
- Configurable output formats including text and images.
- Automatic makefile files generation.
- Trace and operations logs.

Limitations

As other applications, the proposed platform has some limitations that are expected to be solved in future versions:



(a) Square lattice (b) Hexagonal lattice (c) Triangular lattice

Figure 1.3: Tessellations that fill a 2D space. The arrows represents the possible direction of movement to access to the first neighbourhood area, the region made up of the cells located around the central one.

- The number of copies of the sequential algorithm launched by the framework must be defined at the initialisation procedure of the platform. Hence, it is no way of dynamically generate new parallel processes. At the beginning of the execution, the platform launches a fixed number of processes and this number may remain constant during the complete execution of the algorithm on the cluster. Therefore there is no way of using dynamic processes. It will be useful if the platform could decide or adjust the number of processes to fit the size of the problem. In order to do this, the application program will have to interact with the process manager of the MPI layer (commonly called job scheduler) to request and acquire or return computation resources, and with the cluster controller (refer to page 32) in order to take into account the new processes and request its initialisation, communication and global management.
- The procedure of saving the results of the execution of a step of the algorithm must be made by the master process once it receives all the information of all the slave processes; in this way, the master process collects in memory the results of all slaves, before starting the execution of the next step of the simulation. It can then happen that the master process gets out of memory while saving the results, however the results were calculated by the slaves. The alternative is that the slaves, instead of sending the results back to the master, save them directly in local but, this feature has been tested only partially.
- It has been observed that there is a memory leak and sometimes a core dump when using pointers in the definition of the user data object. The solution in this case is to move to static arrays when possible or ANSI C++ containers.

It is worth to mention that the serialisation and deserialisation methods of the object specified by the user to define the values of each vertex of the graph must be provided by the user; this is not really a limitation of the platform while there is no way automatically

determine which fields the object has. Both methods must be complementary. Something similar happens with the method used to read the input values from the input file; as there is no method in C++ to read values "sizeof" the user data and assign field by field the read values, this method must be provided by the user.

1.4.2 Compile

Before compiling the application, this has to be integrated with the platform in order to obtain the generic makefiles and the soft links needed to compile the application. The applications that can run within the platform are, from the point of view of it, organised into schemas or profiles.

The management of the profiles is performed by the batch script `schema` in the following way:

`schema add:` Adds a new application to the platform, collects the compiling information and generates the makefiles files.

`schema del:` Makes an archive of the application and deletes the application from the platform.

`schema archive:` Makes a compressed archived of the platform.

Once the new application has been added to the schema of the platform, it can be compiled by invoking, as usual, the `makefile` form the application root folder or even in the corresponding folder (see chapter 5.4). This `makefile` script compiles the binary file as well as the plugins of the platform implemented by the user and the default ones, and prepares the machine for the execution phase.

1.4.3 Run

The developed application can not be invoked directly by the system as binary files, as they need the MPI process starter. Thus, to launch the execution of any application within the platform it is needed to execute the following line:

This will run *X* copies of `program` in the current run-time environment (if running under a supported resource manager, `mpirun` will usually automatically use the corresponding resource manager process starter, as opposed to, for example, `rsh` or `ssh`, which require the use of a hostfile, or will default to running all *X* copies on the localhost), scheduling (by default) in a round-robin fashion by CPU slot.

```
mpirun [-np X] [--hostfile <filename>] <program> <program
parameters>
```

Frame 1.1: Command line to execute an user application within the proposed platform

The *program parameters* must be always the path to each of the four configuration files that defines the behaviour of the platform as well as the behaviour of the user application and its environment. Refer to chapter 5.3 for more information.

2

State of the Art

In this second chapter of this document, the range of options and procedures performed during the last decades in the techniques of parallel computing, load balancing, partitioning methods is presented. Since this work has been considered in various conferences as a semi automatic parallelisation tool, the last section of this chapter will also provide an introduction to the parallelisation techniques that nowadays exist.

2.1 Parallel Computing

Parallel computing is a form of computation in which many calculations are carried out simultaneously mostly following the principle that large problems can often be divided into smaller ones, which are then solved concurrently [14].

Historically, parallel computing has been considered to be "the edge of computing" [15] and it has been used to solve difficult problems in many areas of science and engineering:

- Atmosphere, Earth and weather problems.
- Physics related problems as applied, nuclear, particle, condensed matter, high pressure... physics.
- Bioscience, biotechnology, genetics...
- Chemistry, molecular sciences, pharmacology...
- Electrical engineering, circuits design, microelectronics...
- Mathematics.
- And in many other fields of science.

Parallel computer programs are more difficult to design than sequential ones [16] because concurrency introduces several new classes of bugs, of which race conditions are the

most common. The communication and synchronisation between the different subtasks are typically some of the greatest obstacles to getting good parallel program performance [17,18].

Traditionally, computer software has been written for serial computation. To solve a problem, an algorithm is constructed and implemented as a serial stream of instructions. These instructions are executed on a central processing unit on one computer (node). Only one instruction may execute at a time; after that instruction is finished, the next is executed.

Parallel computing, on the other hand, uses multiple processing elements simultaneously to solve a problem. This is accomplished by breaking the problem into independent parts so that each processing element can execute its part of the algorithm simultaneously with the others. The processing elements can be diverse and include resources such as a single computer with multiple processors, several networked computers, specialized hardware, or any combination of the above [19].

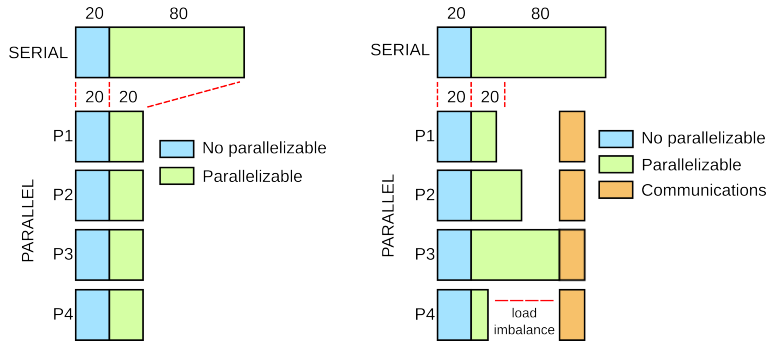
The maximum possible speed-up of a program as a result of parallelisation is obtained though the Amdahl's law. The Amdahl's law is used to find the maximum expected improvement to an overall system when only part of the system is improved. It models the relationship between the expected speed-up of parallelised implementations of an algorithm relative to the serial algorithm, under the assumption that the problem size remains the same when parallelised [20]. The law states that the small portion of the program which can not be parallelised will limit the overall speed-up of the global parallelisation (see figure 2.1 and equation 2.1):

$$1 - P + \frac{P}{(1-P)} \tag{2.1}$$

where P stands for the fraction of code that can be parallelised.

Through the history of multiprocessor systems and from the point of view of workstations and hardware architecture, a variety of parallel architectural models have been proposed. One of the main differences between all these models lies in the sharing of main memory, ie. whether the processors share the same memory for data storage. According to this classification, there are two main families of parallel architectures:

shared memory: Where all processors have access to all memory as global address space. Hence, changes in a memory location done by one processor are visible to all other processors. A shared memory multiprocessor machine can look like a von Neumann machine [21] to which there have been added more processors. All these processors have access to read and write data to the same memory space, accessed through a data bus. All processors work asynchronously and any necessary synchronization mechanism must be done explicitly. All communication, whether data or control



(a) Parallel speed-up: an ideal case. If for example 80% of a program can be parallelized in a system with four processors, the parallel running time will be 0.4, that is, 40% of the serial running time.

(b) Parallel speed-Up: an arbitrary case. Running a parallel program, there is a communication overhead and a workload imbalance involved among processes in general.

Figure 2.1: Amdahl basic concepts. A program solving a large problem will typically consist of parallelizable and non-parallelizable parts.

packets between processors, is performed through the shared resource, the memory. The advantage of shared memory architectures is undoubtedly the fact that communication between processors is very fast. However, they present a major deficiency: the main data bus, common to all processors as a shared resource, can cause a bottleneck problem. Programming models with shared memory focus their interest in asynchronous access to a common resource, the memory address space. Shared memory parallel machines can be subdivided into:

UMA (Uniform Memory Access): Identical processors and equal access and access times to memory. See figure 2.2(a).

NUMA (Non-Uniform Memory Access): Not all processors have equal access time to all memories. Memory access across link is slower. See figure 2.2(b).

distributed memory: In multiprocessor systems with distributed memory, the concept of a global memory, accessible by all processors, does not exist: each processor has its own memory, not visible or accessible by the others. Communication and synchronization between processors, in this case, is done through message passing or tokens. In this architecture there is not a shared data bus, assumed to be a bottleneck. Rather, the communication times are greater than in the case of a shared memory architecture. A particular case of distributed memory multiprocessor are the clusters or group of processors, linked by a network connection. In this particular case, the shared resource is the communication network, being this the main reason to optimize the network topology. Depending on the architectural differences of the computers, homogeneous and

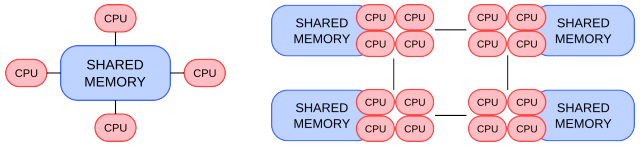
heterogeneous clusters can be found. Heterogeneous clusters [22] have the added difficulty for programming and synchronising the processes. [23]. Distributed memory parallel machines can be subdivided into: [24]

SISD (Single Instruction, Single Data): Only one instruction stream is accessed by a CPU during any one clock cycle and only one data stream can be used as input during any one clock cycle.

MISD (Multiple Instruction, Single Data): Each processing unit operates on the data independently via separate instruction streams and a single data stream is fed into multiple processing units.

SIMD (Single Instruction, Multiple Data): All processing units execute the same instruction at any given clock cycle and each processing unit can operate on a different data element.

MIMD (Multiple Instruction, Multiple Data): Every processor may be executing a different instruction stream and be working with a different data stream.



(a) UMA architecture. (b) NUMA architecture.

Figure 2.2: Shared memory parallel architectures

The programming model for shared memory parallel machines is the thread model [25], where a single process can have multiple, concurrent execution paths. Standardisation efforts have resulted in two very different implementations of threads [26]: POSIX (Portable Operating System Interface) Threads [27] and OpenMP [28]:

POSIX threads: (commonly named *pthreads*) requires parallel coding and provides a very explicit parallelism; it requires significant programmer attention to details. Applications that use *pthreads* are library dependant.

OpenMP: is compiler directive based and can use serial code. OpenMP is portable and multi-platform. It is available in C/C++ and Fortran implementations.

From the point of view of the parallel programming, several models can be found:

Distributed Memory / Message Passing: each task use its own local memory during computation and multiple tasks can reside on the same physical machine and/or across an arbitrary number of machines. Tasks exchange data through communications by send-

ing and receiving messages. This transfer usually requires cooperative operations to be performed by each process. For example, a send operation must have a matching receive operation. The standard interface for message passing implementations is MPI [4].

Data Parallel: Most of the parallel application focuses their features on performing operations on a data set, which is typically organized into a common structure, such as an array or cube. A set of tasks work collectively on the same data structure, however, each task works on a different partition of the same data structure. Tasks perform the same operation on their partition of work. Implementations used for programming this parallel machines are *Fortran90* and HPF (High Performance Fortran) [29,30].

Hybrid: Combination of the message passing model (MPI) with the threads model (OpenMP). An increasingly popular example of a hybrid model is the use of MPI with GPU (Graphics Processing Unit) programming.

SPMD (Single Program Multiple Data): All tasks execute their copy of the same program simultaneously. This program can be threads, message passing, data parallel or hybrid. All tasks may use different data.

MPMD (Multiple Program Multiple Data): Tasks may execute different programs simultaneously. The programs can be threads, message passing, data parallel or hybrid. All tasks may use different data.

Nowadays there are two big branches of great interest to the world of parallel computing, not yet shown in this chapter: *MapReduce* [31] and CUDA (Compute Unified Device Architecture) [32]:

- Hadoop [33,34] is the free implementation of the *MapReduce* technology. Among many parallelizable problems, most applications can be performed using *MapReduce* technologies. It makes sense since infrastructure services as cluster, grid and cloud, allow any user to provision a large number of compute instances fairly easily; but this provisioning of resources happens in minutes as opposed to the hours or even days required in the case of traditional queue-based job scheduling systems. Utilizing these virtual resources to perform data or compute intensive analyses requires employing different parallel runtimes to implement such applications. However, many scientific applications, which have complex communication patterns, still require low latency communication mechanisms and rich set of communication constructs offered by runtimes such as MPI.
- CUDA is the computing engine in Nvidia GPUs accessible to developers through standard programming languages. It gives developers access to the virtual instruction set and memory of the parallel computational elements [35].

2.2 Load balancing

From the point of view of the communications, the main problem related to parallel computing sets out the scenario of designing an interconnection topology in the way that it fulfils the certain features of reliability, assuming the reliability concept as a measure to assess the probability of a successful communication between pairs of nodes, and thus it constitutes a factor of quality of service offered to the different nodes of the cluster.

The evaluation of the exact set of parameters that define the reliability of a network is a NP (Non-deterministic Polynomial time)–Hard problem [36, 37]. The fast development of infrastructures of networks, software design and Internet services has been promoted by the strong demand for data communications over the last 20 years. In that way, the interest on solving network design problems, covering from the optimal placement of antennas, the frequency allocation of mobile telephony and other several structural problems related to routing information through networks has been renewed [38, 39]. As a consequence of the continuous growth in networks capacities, several optimization problems linked to design algorithms have been worldwide proposed.

For a general optimisation of groups of machines [40], a methodology of dynamic resource allocation is needed [41–43] being this, one of the main reasons why today there is particular interest in finding new algorithms, capable of replacing traditional methods whose efficiency and ability to scale in parallel processor architectures make them inapplicable to large problems. In this sense, heuristics have been proposed as methods for solving design problems of reliable communication networks. Although heuristic methods do not guarantee a true optimal value for an optimization problem, they allow to find values quite near to the optimal ones, which its quality may be enough to meet the demands of the designers.

Different parallel programming models differ in certain aspects inherent to the parallel–distributed processing. The main differences concern the level of granularity, the mechanisms used for communication between processes [17], among others:

- The data parallel model is applicable if it is possible to apply a single operation on a multiple set of data. This model produces the lowest level of granularity possible, as it interprets each instructions as an independent task. Generally, it begins with an initial data partitioning over the available processors and with this information, an application program suitable for execution on the SIMD multiprocessors is designed. Parallel libraries such as HPF are based on this model. The model allows to specify parallel programs with a high granularity, from the tasks point of view. Each task encapsulates a sequential program that runs on local memory and are able to send and receive messages, create new tasks and complete its execution. The way two different tasks

communicates each other is based on queues, called channels, which can be created or deleted dynamically.

- On the other hand, the other programming model is the message passing model. It is based on the definition of tasks that communicate each other using explicit primitives through a physical medium. The communication protocol is based on basic operations (primitives) to send and receive messages, allowing to implement data transfers and the synchronization of processes that run even on different processors.

Nowadays there are two programming techniques applicable to message passing model to define the synchronization points between different processors: the technique of functional decomposition and the domain decomposition technique [8]. The functional decomposition technique is based on the identification of modules, that perform independent operations, that could be executed in parallel. This determines that in every instant of time, different code sections or programs could run simultaneously on multiple processors, reducing the global runtime in comparison with the time needed to execute the sequential (not parallelised) application. Within the domain decomposition technique, the parallel application runs, on different processors, the same piece of code, but working concurrently on different datasets. These techniques, the synchronization points, the communication model, as well as the inherent communication delays are the parameters that must be known a priori before designing the parallel algorithm [44].

In order to build a computing environment for farms of computers, it is also necessary to have an algorithm that requires the ability to predict the performance and the resource consumption of different cluster configurations. This algorithm is called the RMS . The problem is that, it is difficult to predict the computing time by a node before it receives some arbitrary load. Also, this will be even more difficult, if we consider the variation of the communication topology of the farm. The basic tasks of the RMS is to accept requests for resources made by the applications and allocate them from the poll of resources. This is a slightly approximation of a computer middleware.

As all nodes are equally master or slaves, we need one special node, the supermaster, in where the RMS plays its role. The RMS uses a predictive estimation based on a mathematical function (heuristic function, Υ), in order to map the tasks of the parallel application to the pool of resources. This heuristic function will be the one needed to define the graph partition and assigning the workload to each node. These techniques for predicting the performance of the dynamic system are nowadays based on queueing techniques and/or on historical techniques [45]. Making a comparison between layered queues and the historical model, it is well known, that the layered queue requires more CPU time to make the mean response time prediction, whereas the historical predictions model are almost instantaneous. However, queuing techniques are easier to implement with a minimum level of performance

than the historical model. This is because designing a historical model involves specifying and validating how predictions will be made, whereas the queueing model can be solved automatically. Both techniques can be combined to take advantages of them [46].

The layered queueing performance model defines an application's queueing network. The solution strategy, in this case, involves dividing the problem into tasks depending on the resources and corresponding them to the tiers of servers in the system model, generating an initial topology solution and then iterating with the historical method, solving and resizing the queues in each step of the algorithm, until the solution converge to an optimal distribution of resources, tasks and communications delays.

For the queue model it is necessary to define a queue structure for each node of the HNoC (Heterogeneous Network of Computers), which will be shared by all the incoming requests. The nodes can be both clients (request information) and servers (process the information). The queue can be managed by a FCFS (First-Come, First-Servered), LCFS (Last-Come, First-Servered) or SIRO (Service In Random Order) policy.

2.3 Domain Decomposition

Graph partitioning is an important problem that has extensive applications in many areas, including scientific computing, VLSI (Very Large Scale Integration) design and task scheduling. The problem is to part the vertices of a graph in roughly equal parts, such that the number of edges connecting vertices in different parts is minimized. DDM (Domain Decomposition Methods) is a branch of mathematics related to numerical analysis used to solve boundary value problems by splitting it into smaller regions.

The problem of partitioning irregular graphs for parallel computations on homogeneous systems has been extensively studied. However, these solutions fail when the target system architecture exhibits heterogeneity in resource characteristics [47].

One of the first steps in designing a parallel program is to know or study how to break the problem into discrete "chunks" of work (commonly called *domains*) that can be distributed to multiple tasks. This action receives the name of "decomposition" or "partitioning".

There are two basic ways to partition computational work among parallel tasks: domain decomposition and functional decomposition:

Domain decomposition: the data associated with a problem is the object that is decomposed. Each parallel task works over a portion of this data.

Functional decomposition: (also called "task parallelism") the focus of this approach resides on the computation that is to be performed rather than on the data to be manipulated or solved by the computation procedure. The problem is then decomposed according to the work that must be done. Hence, each task performs a portion of the overall work from the point of view of the computation. Frequently, the domain decomposition strategy results not to be the most efficient algorithm for a parallel programs. This is the case when the pieces of data assigned to the different processes require greatly different lengths of time to be processed. The performance of the code is then limited by the speed of the slowest process (unless it is not considered a different granularity to different processors). The remaining idle processes do no useful work. In this case, functional decomposition makes more sense than domain decomposition. In task parallelism, the problem is decomposed into a large number of smaller tasks and then, the tasks are assigned to the processors as they become available. Processors that finish quickly are simply assigned more work.

In this work, we will be focused on domain decompositions, leaving the functional decomposition to a future work related to the actual one. For more information about this possible implementation, refer to page 183.

DDM can be subdivided into three big branches:

Spectral partitioning methods: produce good partitions for a wide class of problems, and are used quite extensively [48, 49]. However, these methods are very expensive since they require the computation of the eigenvector corresponding to the second smallest eigenvalue.

Geometric partitioning methods: use the geometric information of the graph to find a good partition. These algorithms [50, 51] tend to be fast but often yield partitions that are worse than those obtained by spectral methods.

Multilevel graph methods: reduce the size of the graph (i.e., coarsen the graph) by collapsing vertices and edges, partition the smaller graph, and then uncoarsen it to construct a partition for the original graph [52, 53]. If very large graphs have to be partitioned, coarsening techniques can reduce the time requirements and therefore the performance of the entire parallel application.

The graph partitioning problem is NP complex, more over, NP–complete. Because of this, many algorithms have been developed and implemented in several graph partitioning libraries such as *Metis* [54, 55] (and its parallel variant *ParMetis* [56]), *Jostle* [57], *Chaco* [58] or *Party* [59]. These libraries usually follow the cited multilevel approach.

The developed framework, focus of this work, uses the geometric partitioning methods, as the spectral methods require high computations and multilevel methods are not as fast as the geometric ones and they also have high needs of resources [60].

When the graph changes over the time (adaptive meshes or adaptive data parallel applications), the computational structure changes from one step to another. Sometimes, if the graph is not repartitioned, it may lead to load imbalance in the time required for computations in each node but, the remapping must have a lower cost relative to the computational cost of executing the iterations for which the computational structure remains fixed. In these cases, the partitioning of the graph needs to be updated: a small number of nodes or vertices of the graph may be added or deleted at any given instant from a partition to another one [61].

2.4 Automatic parallelisation

Designing and developing parallel programs has characteristically been a very manual process. The programmer is typically responsible for both identifying and actually implementing parallelism, taking into account possible bottlenecks, synchronisation points, data transfers, etc, concepts completely away from the sequential code programming. Very often, manually developing parallel codes is a time consuming, complex, error-prone and iterative process.

For a number of years now, several tools have been available to assist the programmer with converting serial (sequential) code into multi-threaded or vectorised (or even both) programs, in order to being able to use multiple processors simultaneously way. The most common type of tool used to automatically parallelise a serial program is a parallelising compiler or pre-processor. A pre-processor generally works in two different ways:

Fully automatic: The compiler analyses the source code and identifies opportunities for parallelism. The analysis includes identifying inhibitors to parallelism and possibly a cost weighting on whether or not the parallelism would actually improve performance. Loops statements are the most frequent target for automatic parallelisation.

Semi automatic: The programmer explicitly tells the compiler how to parallelise the code by using “compiler directives” or “compiler flags”.

Both options are not fully exclusive, since semi automatic parallelisation may be used in conjunction with the fully automatic approach.

The proposed platform follows a semi automatic parallelisation, since the programmer implements the sequential algorithm and the application adds a higher layer to convert this serial code into a parallel one. Hence, it can be considered as a semi automatic parallelisation approach, although from the point of view of the user of the framework, the parallelisation is completely fully automatic.

I

METHODOLOGY

3 Architecture and Design

The framework has been designed and implemented following a layered model: a way of dividing the complexity of the problem to be solved in parallel into more and more simple objects, until the physical layer, needed to communicate the processes, is reached. The even more simple objects are sent through the net by using the most lower layer, the MPI, from one process to another one (point-to-point communications), to a group of processes (group communications) or to all processes (broadcast communications).

In the case on review now, each layer of the proposed framework communicates only with its upper and lower layer but it will never ask for, or return information to its second upper or lower layer. This condition minimize the complexity of the platform on its own and reduce the efforts to test the functionality and features of the application.

In following sections, the architecture and the design of the platform will be explained. For a deeper knowledge, see the appendix **??**.

3.1 Architecture

The architecture of the framework is, in essence, divided into four main blocks (see figure 3.1):

- Cluster controller.
- master process controller.
- slave processes controller.
- MPI Layer.

Both in master and slave processes controller there are several pure virtual classes that could be overwritten to allow a generic behaviour of the framework from the point of view

of the loaded plugins, algorithm and several objects that the user must implement. These object, plugins and algorithm are explained in the next sections.

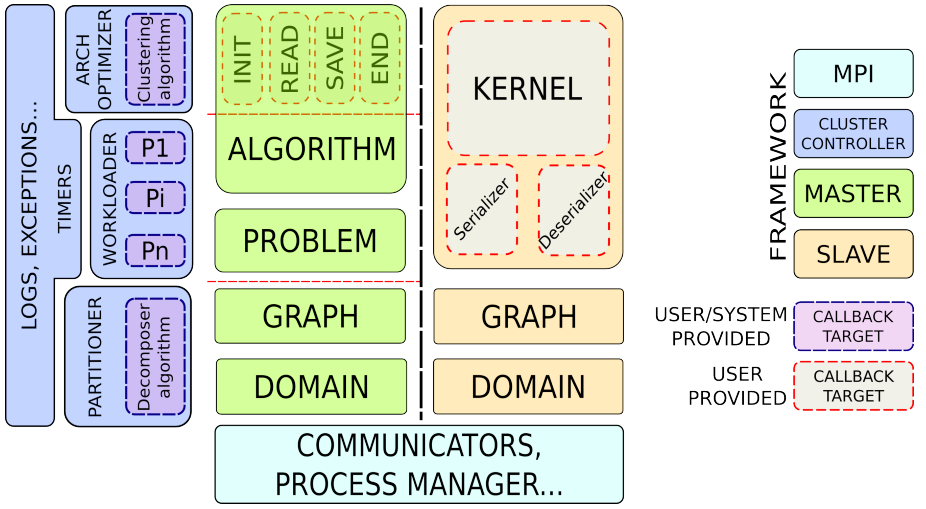


Figure 3.1: Architecture of the proposed framework. There are four main control blocks: the cluster controller, the master process controller and the slave processes controller and the MPI communication layer.

The framework has grown using the features of the *Message Passing Interface* (MPI, see appendix A). Although the MPI Layer is part of the framework, it has not been developed for this project.

3.1.1 Cluster controller

This big block is in charge of managing the cluster as a collection of processes. It consists of four modules: the architecture optimizer, for minimizing the overall cost of sending information between machines of the cluster; the load balancer, in charge of assigning the number of tasks for each slave according to certain characteristics measured in each node; the partitioning algorithm, which decomposes the problem into independent sub-domains; and as last module, the notifications handler, responsible for catching the exceptions, controlling errors and starting and stopping timers for the loadbalancer.

These four modules are only accessible from the master process avoiding the user to manage them and to deal with the exceptions thrown by the framework in case of error. It also avoids the manipulation of the timers to the users.

Architecture optimizer

Since the cluster can be considered a graph, it may happen that more than one path between two nodes exists, possibly with different total cost. Therefore it is necessary to find an optimal path between each pair of nodes. Such operation can be performed using algorithms such as *Dijkstra* or even by a classification algorithm based on a set of parameters.

As the optimizer can vary between two different simulations or even vary to test the performance of the problem to solve, the user can load its own algorithms to optimize the resulting system graph. For an extended information about how the user can modify the way the platform creates clusters of information, refer to page 54.

Load balancer

This module is the one responsible for distributing the processing load evenly among all slave processors, ie. it specifies the number of tasks (granularity) assigned to each slave, as uniform as possible (see page 9) taking into account several features of each node.

Different factors can be taken into account to estimate the load balancing and granularity per slave, depending on the problem to solve. Thus, the user can implement and load its own load balancing plugins to optimize the tasks distribution feature of the platform (see expression 4.1).

Domain decomposer

The partitioning algorithm takes care of, once the conditions of the load balancer and the number of slaves involved in the computation are known, determining the set of partitions (domains) in which the problem must be split, so that either the number of communications that connect two partitions (communications outside the domain) are minimized, or the compression of the partitions is maximized. Although the framework itself contains three partitioning algorithms which can be used by the user, the framework allows the user to implement his own partitioning algorithms and to use them by loading them into the system (see pages 56 and 104 and the section 4.2 to understand how can the plugin be specified and what it really means).

Handlers

A fourth module, handlers, is responsible for both, monitoring the actions taken by the cluster and managing the timers of the system which are needed for example to the time-based load balancer plugins, as for example the historical plugin, to calculate the RMS values (see

expression 4.1 and pages 23 and 56).

The behaviour of this module can be modified by changing between the different debugging levels available for the user. This levels are correspond to the values assigned to the environment variable `$DEBUG`.

At this level, the exceptions that may occur during the execution are managed and, depending on the severity of them, caught as well as the ones related to the control of the internal errors that may also crash the simulations.

3.1.2 Master process controller

The master process, whose MPI identifier is equal to zero, is launched in the "local" node. The remaining processes may be both on the same machine or distributed, along the cluster, but anyway there will have always a greater identifier.

Basically, the master process plays the role of a conductor ie. reserves and initializes the necessary structures (except those used specifically by the slaves), transforms the problem defined by the user into a multidimensional application graph, estimates the loading rates of each node, decomposes the problem into several domains, sends the tasks, controls the execution of the slave processes, gathers the results returned by other processes and reconstructs the original problem with the recently received results. In other words, it synchronizes the processes of the entire execution.

Users should implement at this level some functions, which will be called as callbacks from the master core.

3.1.3 Slave processes controller

The execution of the user defined algorithm in the cluster is managed by the set of slave processes. These processes do not have information about the cluster and the problem to be solved; they only receive a graph object, part of the complete application graph, and apply the kernel implemented by the user, returning the results back to the master process. Neither they know how many other slaves are involved in the computations or even if what they have received is the complete graph of the problem or if it is a sub-graph.

At this level the users that want to use the framework must implement some callback functions which will be called by the slave process controller, such as for example the way the user defined object is serialized or how the simulation/execution will carry out.

3.1.4 MPI Layer

MPI is commonly defined as: “message passing application programmer interface, together with protocol and semantic specifications for how its features must behave in any implementation” [62].

MPI is an API specification that allows processes to communicate with others by sending and receiving messages. Besides many other applications, it is a *de facto* standard for parallel programs running on computer clusters and supercomputers, where the cost of accessing non-local memory is high. Its goals are high performance, scalability and portability.

Although MPI belongs in layers 5 and higher of the OSI (Open System Interconnection) reference model, implementations may cover most layers, with sockets and TCP used in the transport layer. Both point-to-point and collective communication are supported by MPI API and therefore by the framework.

MPI uses LIS (Language Independent Specifications) for calls and language bindings and it is directly callable from C, C++, Fortran and any language able to interface with such libraries, such as C#, Java or Python

This layer is the only one that has not been developed nor changed for the actual work.

3.2 Design and Implementation

The proposed framework is implemented in ANSI C++ language, uses the open MPI library on its 2.0 version and *protobuffers* version 2.4.1 (but it is compatible with lower versions of branch 2 through recompiling the framework application) for the management of configuration files. Protocol buffers (see appendix B) are a serialisation format with an interface description language developed by *Google*. The original *Google* implementation for C++, Java and Python is available under a free software, open source license. The framework runs under any GNU/Linux flavour that can run and compile the cited dependencies.

The design of the platforms follows a modular design, concept also called “modularity in design”. This model is an approach that subdivides a system into smaller parts (modules) that can be independently created and then used in different systems, to drive multiple functionalities. Each one accomplishes basically one function and contains everything necessary to accomplish it.

In the next subsections, the namespaces, main classes and interfaces are going to be introduced one by one. A more detailed information can be found on appendix **??**.

3.2.1 Classes

Several classes have been developed; each one represents a general concept, from the point of view of the implementation of the framework as a whole.

The aim of this section is not to list all the classes (code construction that consists of and is composed from structural and also behavioural constituents and can be instanced) that have been designed and implemented but, in order to understand the next sections and chapter, it may be useful to at least take the following ones into account.

Configuration

The `Configuration` class (codes 3.1) is the father class of all the configuration files that the user need to write before launching the platform. It defines a class with all the parameters that the user can define to modify the behaviour of the framework.

This class is directly connected to the *protocol buffers* library, the one that parsers the user configuration files.

```
21      class Configuration
22      {
23          public:
24              std::string sName;
25              unsigned int iNodes;
26              unsigned int iLinks;
27              std::string *node_sName;
28              double *node_dCPU;
29              double *node_dMemory;
30              std::string *link_sSource;
31              std::string *link_sDestination;
32              double *link_dWeight;
33
34              std::string heuristic_sName;
35              int heuristic_iSteps;
36              double heuristic_dAcceptance;
37              int heuristic_iIncrement;
38
39              unsigned int iPlugins;
40              unsigned int iDepth;
41              std::string *plugin_sName;
42              double *plugin_dWeight;
43              std::string sMethod;
44
45              unsigned int iConnection;
46              std::map<std::string, Reconfiguration::Coordinate> mCells;
47              std::map<std::string, int> mLinks;
48              Vector<std::pair <int, std::string> > vConnections;
49
50              std::string simulation_sName;
51              int simulation_iDimension;
52              int *simulation_iDimensions;
53
54              double *simulation_dHistorical;
55              bool *simulation_bBoundaries;
56              std::string simulation_sInput;
57              std::string simulation_sTime;
58              std::string simulation_sLog;
59
60              int simulation_iHow;
61              bool simulation_bSavePartitions;
62              std::string *simulation_sOutput;
63              std::string *simulation_sFormat;
64
65              int *iAccDimensions;
66              int iFiles;
67
68              Configuration();
69              Configuration(std::string);
70              virtual ~Configuration();
71
72              virtual void read()=0;
73              virtual void write()=0;
74              virtual void print()=0;
75      };
```

Code 3.1: Configuration class definition.

Communicator

The `Communicator` class (code 3.2) was developed with the aim of improving the functionality of the MPI layer by adding new functions to the standard ones, needed for the framework or even to facilitate the management and calls to the standard MPI library. This class can be defined as a middle layer inbetween the MPI layer.

```
10  class Communicator
11  {
12      public:
13          MPI_Comm handler;
14          Communicator();
15          Communicator(MPI_Comm);
16          virtual ~Communicator();
17          int probe (int, int*, int*, MPI_Status*);
18          int getCount(MPI_Status, MPI_Datatype, int *);
19  };
```

Code 3.2: Communicator class definition.

Framework

The `Framework` class (code 3.3) initializes the MPI layer also with its error handlers procedures, as well as it creates the master and slave processes and manage them from a very upper point of view. All processes, regardless of its type (master code 3.10 or slave code 3.9), must join themselves to the execution launched by the user before they can send or receive any network packet from any other process.

```
 8  class Framework
 9  {
10      public:
11          Framework(int, char **, Algorithm *);
12          virtual ~Framework();
13  };
```

Code 3.3: Framework class definition.

The framework module acts as the main module, joining all the implemented ones and communicates directly with the main method of the user defined code.

Graph

The `Graph` class (code 3.4), as its name specifies, allocates a graph that defines the user defined problem and provides an API to manage this structure. Conceptually, the graph is nothing more than a collection of vertices (see code 3.12) although the class includes other fields in order to access to its elements in a more efficient way.

The problem and its data can be considered as a graph. A graph structure is the most generic one, where any other structure can be easily mapped. To manage, from a transparent point of view, the different dimensions the graph may have, it must be flattened into a simple one dimensional vector where the indexes are mapped in an unique way, to avoid indexes collisions or out of bounds errors. This way, advantages of continuity of memory can be taken.

Having a N-dimensional graph, identified by G_N, where its vector of sizes per dimension is:

$$\boxed{V_N = \{D_N, D_{N-1}, \cdots, D_1\}} \tag{3.1}$$

being D_i the number of elements of the graph G_N at dimension i, an array of accumulated offsets per dimension can be obtained as:

$$\boxed{Acc_{N+1} = \left(\prod_{i=1}^{N} D_i, \prod_{i=1}^{N-1} D_i, \cdots, D_2 \cdot D_1, D_1, 1\right)} \tag{3.2}$$

The procedure to obtain the lineal offset from a multidimensional position is computationally very simple and involves, in this way, both vectors:

$$\boxed{P = Acc_N V_N^T} \tag{3.3}$$

where Acc_N is the vector of the N last elements of Acc_{N+1} and V_N has been transposed just to apply a vectorial multiplication of both vectors. (see figure 3.2).

The conversion between indices is done when the graph and the neighbourhood of each of its vertices are defined. The user can then access to the elements of the graph as if it were a matrix, that is, specifying the offsets per dimension. The operator for accessing to the items of the graph (operator [ ]) has been overridden in order to present to the user a completely transparent way to access to the elements. This procedure involves several fields of both the `Graph` class as the `Problem` class, as also the description of the directions defined by the user in the configuration file (see code 5.11):

- The *Configuration* class (see code 3.1) stores the `mCells` field that allocates a map structure with the name given by the user in the configuration file for a specific direction and also the Cartesian coordinates that defines each direction (see figure 3.3(a)).

```
62      class Graph
63      {
64          public:
65              int iDimension;
66              int *iAccDimensions;
67              Vector<Vertex> vVertex;
68              std::map<int, int> mHash;
69              std::map<std::string, int> mLinks;
70
71              Graph();
72              Graph(std::map<std::string, int>, int, int*);
73              Graph(const Graph &);
74              virtual ~Graph();
75
76              Data operator()(int) const;
77              Data& operator()(int);
78              Data operator()(int, const std::string&);
79              Data operator()(int, int*);
80
81              int idVertex(const std::string&);
82              int idVertex(int);
83
84              void addVertex(int, double, double, const std::string&);
85              void addVertex(int, Data);
86              void addVertex(const Vertex&);
87
88              void setVertexValue(int, const Data&);
89              void setRMS(Vector<double>);
90
91              void addLinks(const std::string&, const std::string&, double);
92              void addLinks(int, int, double);
93
94              double searchLink(int, int);
95              Data* searchNode(int, double);
96
97              int size();
98              Vector<int> elements();
99              Vector<Data> neighborhood(int);
100
101             void print();
102     };
```

Code 3.4: Graph class definition.

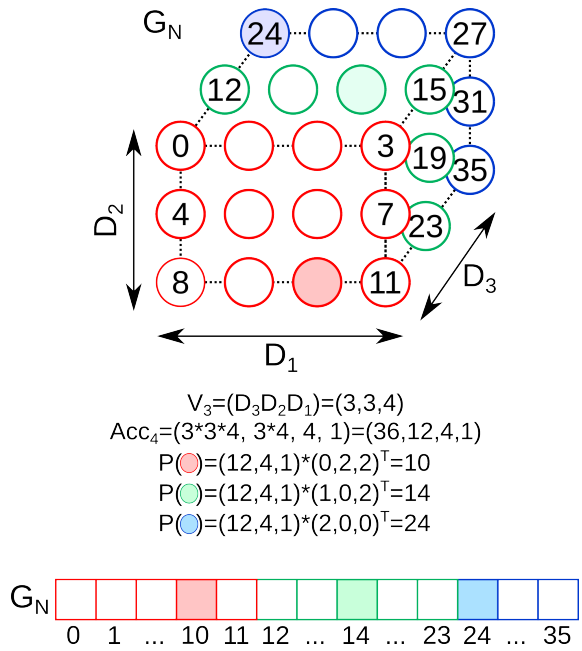


Figure 3.2: Flattening a multidimensional graph into an unidimensional vector.

- The *Graph* class, as shown in code 3.4, defines the `mLinks` map structure, which joins the name of the direction previously defined and the lineal offset of that direction (see figure 3.3(b)).
- The *Problem* class (see code 3.7) defines the `vConnections` vector, in order to group the vertices of the graph with the directions of their neighbours (see figure 3.3(c)).

Once the user specifies either the name of the direction defined at the configuration file or directly the offsets per direction, to determine the neighbour that linked with the actual one fulfilling the direction, the next steps are carried out:

1.– The given name of the direction and the offsets have to be stored at the `mCells` field of the *Configuration* class.

2.– Once the graph is instanced, the directions of the `mCells` field are translated into lineal offsets by applying the 3.2 expression and stored at the `mLinks` field of the *Graph* class.

3.– The direction must be a valid direction of the actual vertex, checking this at the `vConnections` field of the *Problem* class.

4.– The second field of the `mLink` field is taken and added to the index of the actual vertex, resulting the target vertex of the direction.

"E"	(1,0)
"S"	(0,1)
...	...

(a) mCells field which joins the given name identifying the direction and the Cartesian coordinates that define it.

"E"	+1
"S"	+4
...	...

(b) mLinks field joining the direction name and the final lineal offset once the graph has been flattened.

0	"E"
0	"S"
1	"E"
1	"S"
...	...

(c) vConnections makes a relation between the vertices of the graph and the direction of their links.

Figure 3.3: The three object fields involved in the process of accessing to an item of the graph object.

Output

In every C++ program (or library, or object file), all non-static functions are represented in the binary file as symbols. These symbols are in special text strings that uniquely identify a function in the program (or library, or object file). Because C++ allows overloading (different functions with the same name but different arguments) and has many features C does not — like classes, member functions, exception specifications — it is not possible to simply use the symbol name as the function name. To solve that, C++ uses something called *name mangling*, which transforms the function name and all the necessary information (like the number and size of the arguments) into something like a string which only the compiler knows about. Thus, C++ language does not have a feature to instance objects from their name, as for example JAVA or C# have, so it is not possible to create objects, in this case in charge of saving the information, directly in runtime.

As the user plugins are not generally provided by the framework, the framework has any knowledge of their existence, name or even structure. In this way, all plugins must inherit from a special class that only provides the definitions to the framework (class commonly named as *interface*; see page 54), specifications that all plugins must fulfil to communicate with the framework. Since the user specifies the name of the plugin or plugins that wants to use, to create the corresponding object, they must be defined somehow in a file which its name is related to the name of the plugin and that can also be in loaded in runtime; that is, as a dynamic library. Dynamic linking involves loading the subroutines of a library into an application program at load time or runtime, rather than linking them in at compile time. Once the library is loaded, asking for the interface structure, the special symbol *new* can be called and thus, the user-defined class can be instanced in runtime.

```
13      class Output
14      {
15          Vector<void*> vLibrary;
16          Vector<IFormat*> vFormat;
17          int iFiles;
18
19          public:
20              Output(int, std::string*, std::string, int, int*, bool);
21              virtual ~Output();
22              void save(std::stringstream*, bool);
23      };
```

Code 3.5: Output class definition.

The constructor and destructor methods can be only called by symbols, because interfaces can not be instanced; once the *new* symbol of the library is called, the new object can be cast to the interface type object, allowing the normal invocation of methods.

In the `Output` class (code 3.5), the communication with the output saving plugins is done, that is, the management of the user plugins that save the results of each step of the simulation into a file, whatever this a text-file or image-file is.

Partition

The `Partition` class (code 3.6) represents and idealizes a group of neighbouring vertices that make up each domain in which the problem has been parted into.

slave processes need to know not only the domain to apply to, but also the data associated to that domain and thus, the master process has to being able to serialize the block that represents each partition.

The `Partition` class contains control information and data information (see expression 3.4c):

- The control information (see expression 3.4a) contains how many nodes, N, are part of the graph (sub-graph) that represents the partition; which vertices are involved ids, identified by their identifier value; the index they occupy in the vector of vertices, $index$, (see image 3.2); the edge of those vertices, $edges$ and a status value, $status$, which means if that value is part of the partition `RW (Read write)` or if it is only a dependency of other vertex `RO (Read only)`.
- The data information (see expression 3.4b) contains the array of values of each vertex, $values$.

```
29      class Partition
30      {
31          public:
32              int iNodes;
33              Data *values;
34              int *iIds;
35              int *iIndex;
36              int *iEdges;
37              int *iLinks;
38              int *iStatus;
39
40              int iControlLength;
41              int iDataLength;
42
43              Partition();
44              Partition(int, int);
45              Partition(const Partition&);
46              virtual ~Partition();
47
48              double* encapsulate ();
49              char* encapsulateData ();
50              void desencapsulate (double *);
51              void desencapsulate (char *, int);
52
53              void partition2graph(Graph *);
54              void graph2partition(Graph *);
55
56              void print();
57      };
```

Code 3.6: Partition class definition.

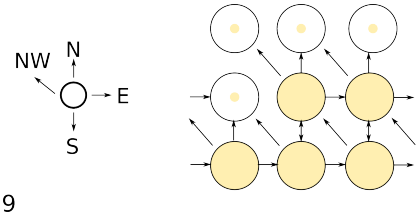
Taking this into account, the buffer where the partition is dumped occupies:

$$C = (1 + 3N + 2L)sizeof(INT) \tag{3.4a}$$

$$D = Nsizeof(UserData) \tag{3.4b}$$

$$Csizeof(CHAR) + D \tag{3.4c}$$

bytes, where N is the number of nodes of the partition, L represents the number of total links in it, `sizeof(INT)` and `sizeof(CHAR)` depends on hardware architecture and `sizeof(UserData)` depends on the user datatype allocated on each vertex of the graph. The *1+* component of the expression refers to the integer that specifies the number of nodes of the partition (cardinality); $3N$ to the cardinality of the ids, $index$ and $status$ arrays and $2L$ to the cardinality of the $edges$ and $links$ arrays.



nodes=9
id={0,1,2,3,4,5,6,7,8}
values={(...)}
index={0,0,0,0,4,8,11,14,17}
edges={1,5,7,0,2,3,8,1,3,7,5,4,8,3,5,6,4}
links={NW,N,E,S,NW,N,E,S,NW,N,E,NW,N,E,NW,N,E}
status={RO,RO,RO,RO,RW,RW,RW,RW,RW}

Figure 3.4: The nodes are numbered from 0 to $N-1$. The i entry of array index stores the total number of neighbours of the first i graph nodes. The lists of neighbours of nodes $0, 1, ..., N-1$ are stored in consecutive locations in array edges. The total number of entries in index is N and the total number of entries in edges is equal to the number of graph edges. $index[0]$ is the degree of node zero and $index[i] - index[i-1]$ is the degree of node $i, \forall i = 1, ..., N-1$. The list of neighbours of node zero is stored in $edges[0]$ and the list of neighbours of node $i, ..., N-1$ is stored in $edges[j]$. The values are serialized using the serialisation method implemented by the user. The status of each vertex of the partition graph can be `RW` if this value is going to be recalculated or `RO` if it is a dependency of a `RW` vertex. White circles with a point inside identify the `RO` nodes. As these nodes are not updated, they do not have out links connecting to other nodes.

As the data associated to each vertex of the graph is an user-defined object that initially is not part of the framework, it is not possible to automatically serialize and deserialise this field of the partition. As a consequence of this, the user must implement two methods to

enable communication with the framework, so that the complete definition of the partition can be done.

Following the recommendations made by the "MPI Forum" [63] for the serialization of graphs, some of those ideas were here applied for the serialisation and representation of each domain (see figure 3.4 for a graphical example).

Problem

The problem class (code 3.7) allocates the information of the problem that is going to be simulated/solved on the cluster using the proposed platform. It also acts as an integration module to join the concepts of graph, partitions and neighbourhood (topology of the problem) at the master side.

```
59      class Problem
60      {
61          private:
62              int iLength;
63              void *library;
64
65          public:
66              int iPartitions;
67              int iDimension;
68              int *iAccDimensions;
69
70              Graph *data;
71              Vector<std::pair<int, std::string> > vConnections;
72              std::map<std::string, int> mLinks;
73              Vector<Partition*> vPartitions;
74              ISorting *sorting;
75
76              Problem();
77              Problem(int, int, int*, std::map<std::string, Coordinate>,
                    std::map<std::string, int>, Vector<std::pair<int,std::string> >,
                    bool*, std::string, std::string, Algorithm*);
78              virtual ~Problem();
79
80              void createGraph(bool*, std::map<std::string, Coordinate>, int);
81              void part(Vector<int>);
82              void gather();
83
84              int sizeOf();
85              void print();
86      };
```

Code 3.7: Problem class definition.

This structure is only shared by the master processes and it is completely unknown for the slaves, as they do not need to know nor the topology of the problem neither the entire graph.

RMS

The RMS class (code 3.8) is mainly responsible for the load balancing of the slaves involved in the execution and it acts as the manager of all the load balancing plugins that the user has specified in the correspondent configuration file.

As in the case of the output saving plugins (see page 42), the load balancing plugins are implemented as dynamic libraries, being the RMS module the one that manages and allows the communication between the interface that defines the plugins and the framework as a whole entity.

The RMS module integrates also the timers (see code 3.11), as this ones are needed in case the load balancing is based on historical or recent timestamp values.

```
16      class RMS
17      {
18          int iDepth;
19          double *dHistoricalWeights;
20          double *dWeights;
21          int iNodes;
22
23          Graph *graph;
24          Vector<IPlugin*> vPlugins;
25          Vector<void *> vLibrary;
26
27          public:
28              Vector<double> vRMS;
29              Timer *timer;
30
31              RMS();
32              RMS(Graph*, int, int, std::string *, double*, double*, std::string);
33              virtual ~RMS();
34
35              void get(int);
36              void print();
37      };
```

Code 3.8: Problem class definition.

Slave

Slaves processes are independent entities that solve a part of the application by calling an user defined method where the behaviour of the slave is defined. Joining the results obtained by all the slaves, the solution of the global application can be obtained.

The `Slave` class (code 3.9) defines the behaviour of the slave processes once they have been created in runtime as a system process by the MPI library. From the point of view of

the MPI layer, slave processes are those with a non zero identifier.

```
91      class Slave : public Process
92      {
93          private:
94              int iMaster;
95              Partition *partition;
96              std::map<std::string, int> mLinks;
97
98          public:
99              int iDimension;
100             int *iAccDimensions;
101
102             Slave();
103             Slave(int, int, Algorithm *);
104             virtual ~Slave();
105
106             void join();
107             void run();
108             void send(int);
109             void receive();
110             void receivePartition(int);
111             void receiveData(int);
112             void end();
113     };
```

Code 3.9: Slave class definition.

SuperMaster

The `Supermaster` class (code 3.10) allocates the master process and its functionality. The master process is created by the framework once the user invokes the application and receives always the zero identifier. Its procedure involves the creation of all the structures needed for the simulation, from the point of view of the master execution, that is, the management of the cluster topology, the creation of the problem object with the graph and the partitions, the interfaces to manage the user plugins and the structures to manage the results and the answers of the slave processes.

The only interaction that exists between the framework and the user, is made by the supermaster process.

Timer

In the `Timer` class (code 3.11) resides the logic to measure the elapsed time between different events. This time interval is always measured in milliseconds.

Timers are used by the RMS module (see page 47) in case the user specifies the use of

```
47      class SuperMaster : public Process
48      {
49          private:
50              int iSlaves;
51              int iLength;
52
53              std::string sName;
54              std::string sPartition;
55
56              int iReconfSteps;
57              double dAcceptance;
58              int iIncrement;
59
60              Vector<bool> vFinished;
61              RMS *rms;
62
63              Graph *cluster;
64              Problem *problem;
65              Output *results;
66              Timer *timer;
67
68              std::stringstream *ss;
69              int iFiles;
70              int iHow;
71              bool bSavePartition;
72
73              fstream log;
74              int isFinished();
75
76              void sendPartition(int, double*);
77              void sendData(int, char*);
78
79          public:
80              SuperMaster();
81              SuperMaster(char **, int, Algorithm *);
82              virtual ~SuperMaster();
83
84              void join();
85              void run();
86              void send(int);
87              void receive();
88              void end();
89      };
```

Code 3.10: Supermaster class definition.

```
13      class Timer
14      {
15          public:
16              std::fstream output;
17              Vector <double> vTime;
18              Vector <double> vHistorical;
19
20              Timer();
21              Timer(const char*, int, int);
22              virtual ~Timer();
23
24              double get (int);
25              void up(int);
26              void down(int);
27              void write (int);
28      };
```

Code 3.11: Timer class definition.

load balancing plugins based on recent or historical timestamps. Even if the load balancing policy is not based on time plugins, timers will be used if the framework is executed with the verbose debugging output, specified by the highest value of the environmental variable `$DEBUG`.

Vertex

The graph shown on page 40 is a unidimensional collection of objects of type `Vertex` (see code 3.12).

From the point of view of the platform, there are two kind of vertices and thus, two different possible graphs, with the same definition but corresponding to two different concepts, application graphs and system graphs:

System graphs: commonly referred as *clusters*, identify the network topology where the application is running. (see expression 1.1)

Application graphs: represent the dataset of the problem that the user defines. (see expression 1.4)

In this way, the system graph is a collection of vertices that allocates the information of the nodes of the cluster (class `Machine`, not shown at this point) and the application graph allocates the information of the user defined problem (class `Data`, not shown in this section neither), that is, the values of the problem that the user specifies.

Both kind of vertices stores the information of which links that vertex has, regardless of

```
40      class Vertex
41      {
42          public:
43              int iId;
44              Machine *machine;
45
46              Data *value;
47              int iType;
48              int iLinks;
49              std::list<Link> vLinks;
50
51              Vertex();
52              Vertex(const Vertex&);
53              Vertex(int, double, double, const std::string&);
54
55              Vertex(int, const Data&);
56              virtual ~Vertex();
57
58              Vector<int> neighborhood();
59              void print();
60      };
```

Code 3.12: Vertex class definition.

the vertex is part of an application graph or a system graph.

3.2.2 Abstract classes

An abstract class is designed only as a parent class from which child classes may be derived. They can be defined as super-classes which contain abstract methods; subclasses extend them by implementing these abstract methods. The behaviours defined by such a class are "generic" and much of the class will be undefined and unimplemented. Before a class derived from an abstract class can become concrete (can be instanced), it must implement particular methods (may or may not be all of them) for all the abstract methods of its parent classes.

The two most important implemented abstract classes are the ones specified in the following sections: algorithms, exceptions and processes.

Algorithm

As the problems that users want to solve using the platform are not really part of it, they must have a way to communicate with the framework, but to communicate with the framework, all of the user defined problems must fulfil a set of conditions. This communications and the conditions to allow both side to understand each other is provided by the `algorithm` abstract class (code 3.13).

```
12      class Algorithm
13      {
14          public:
15              int iRank;
16
17              Algorithm();
18              virtual ~Algorithm();
19
20              virtual int process (const char*, Data**, int)=0;
21              virtual void kernel(Graph**)=0;
22              virtual void save(Graph*, std::stringstream*)=0;
23              virtual int end(Graph*)=0;
24
25              void set(int);
26              virtual void postGather(Graph **);
27      };
```

Code 3.13: Algorithm abstract class definition.

This class provides the methodology to let the framework understand, for example, the data object defined by the user, how are they read from an input file, how must the results be saved into an output file, and the most important method, how the simulation carries out and when is it finished.

baseException

The `baseException` abstract class (code 3.14) is the father of all the exceptions that the framework can throw.

```
10      class baseException
11      {
12          std::string sMessage;
13
14          public:
15              baseException();
16              baseException(std::string);
17              virtual ~baseException();
18
19              void print();
20      };
```

Code 3.14: Exception abstract class definition.

The use of only one father exception class provides transparency to the upper classes of the framework. All exceptions from the platform inherits from this base class, overwriting its catching behaviour according to the severity of the exception and the actual scenario. The

exceptions classes that inherit form `baseException` are:

`ConfigurationException:` thrown if a configuration error occurs.

`InternalException:` thrown if an internal error occurs.

`InvalidArgumentsException:` thrown if a function receives an invalid argument from an upper layer or method.

`MPIException:` thrown if an error in the MPI layer occurs.

`NoSuchFileException:` if a non-existent file is requested.

`OutOfBoundException:` thrown if the limit of an structure is exceeded due to an invalid conversion between the multidimensional and the lineal offsets.

Process

The `Process` abstract class (code 3.15) is therefore responsible for the management of the processes and for the creation, join, execution and other actions of the different processes.

```
26      class Process
27      {
28          public:
29              int iRank;
30              int iSize;
31
32              Communicator *communicator;
33              Algorithm *algorithm;
34
35              Process();
36              Process(int, Algorithm *);
37              Process(int, Algorithm *, int);
38              virtual ~Process();
39
40              virtual void join()=0;
41              virtual void run()=0;
42              virtual void send(int)=0;
43              virtual void receive()=0;
44              virtual void end()=0;
45      };
```

Code 3.15: Algorithm abstract class definition.

Although at the moment only processes with the category of master or slave have been introduced, there is still a third type of processes, the supermaster processes.

As the system graph must be mapped into the cluster topology, this one can be designed as a $n - tier$ architecture, where more than one level of abstraction exist. In this case, slave processes, called for instance A, are defined as those that receive the partition with the data

values from a parent node B, compute the solution of the algorithm specified by the user applied to the received partition and send back the results to the same parent process. This parent process B can be also a slave process from another one, called for example in this case C. In this way, the node B is a master process from the point of view of the nodes A but a slave process for the C one. Iterating this hierarchy, there must exist a special process parent of all the other ones that only acts as a master, process that in the platform receives the name of supermaster node.

3.2.3 Interfaces

The named modules of the design of the framework are typically incorporated into the program through interfaces, pure virtual classes that provide required information to modules from a transparent point of view. The interface of a software module A keeps deliberately separate from the implementation of that module. The interface contains the actual definition of the procedures and methods described in the module, as well as other "private" variables, procedures and some other possible elements. Any other software module B (which can be referred to as a “child” of A or of “type” A) that interacts with A, is forced to do so only through the interface.

One practical advantage of this arrangement is that replacing the implementation of A by another one that meets the same specifications of the interface, should not cause B to fail or not compile, providing a cleaner and a more maintainable code, and a more reliable continuity across versions.

The most relevant interfaces developed to implement the functionality of the framework can be seen in the following sections and pages.

IFormat

The interface `IFormat` (code 3.16) allows the users to load their own output plugins to save the results of the executed algorithm.

IGrouping

The interface `IGrouping` (code 3.17) lets the users to use their own algorithms to optimize the total costs of the paths that joins the nodes of the system graph (see definition at page 50) between them (see page 33).

```
10      class IFormat
11      {
12          public:
13              FILE *oFile;
14              std::string sName;
15              int *iDimensions;
16              int iStep;
17
18              IFormat();
19              virtual ~IFormat();
20
21              void set(std::string, int, int*);
22              virtual void save (std::stringstream *)=0;
23      };
24
25      typedef IFormat* formatcreate_t();
26      typedef void formatdestroy_t(IFormat*);
```

Code 3.16: IFormat pure virtual class definition.

```
45      class IGrouping
46      {
47          public:
48              int iGroups;
49              Vector<Graph> vGraph;
50
51              IGrouping();
52              IGrouping(int);
53              virtual ~IGrouping();
54
55              virtual void algorithm(Graph *)=0;
56      };
57
58      typedef IGrouping* groupincreate_t();
59      typedef void groupingdestroy_t(IGrouping*);
```

Code 3.17: IGrouping pure virtual class definition.

IPlugin

The interface `IPlugin` (code 3.18) allows using different loadbalacing plugins in order to equally distribute the load between parallel processes.

```
24      class IPlugin
25      {
26          public:
27              Vector<double> v;
28              std::string sName;
29              double dWeight;
30              double *dHistoricalWeights;
31              int iDepth;
32              Graph *graph;
33              Timer *timer;
34
35              IPlugin();
36              virtual ~IPlugin();
37
38              void set(std::string, double, int, double*, Graph*, Timer*);
39              virtual void get()=0;
40      };
41
42      typedef IPlugin* plugincreate_t();
43      typedef void plugindestroy_t(IPlugin*);
```

Code 3.18: IPlugin pure virtual class definition.

ISerializable

As the values of the problem is a user defined class, the framework can not understand which datatypes are inside this class and therefore it is necessary to specify different methods to communicate these data objects with the framework by serializing and deserializing them. Both methods are defined at the `ISerializable` interface (code 3.19).

```
21      class ISerializable
22      {
23          public:
24              ISerializable();
25              virtual ~ISerializable();
26
27              virtual std::stringstream& operator<<(std::stringstream&)=0;
28              virtual std::stringstream& operator>>(std::stringstream&)=0;
29      };
```

Code 3.19: ISerializable pure virtual class definition.

ISorting

The way the problem is parted into different and independent domains depends on an algorithm that could be also implemented and loaded by the users and may not be part of the platform. To allow the communication between this algorithm and the platform, these algorithms must inherit from the special interface `ISorting` (code 3.20).

```
19  class ISorting
20  {
21      public:
22          int iSize;
23          int *iSort;
24
25          ISorting();
26          virtual ~ISorting();
27
28          virtual void sort(Graph *)=0;
29          void print();
30  };
31
32  typedef ISorting* sortcreate_t();
33  typedef void sortdestroy_t(ISorting*);
```

Code 3.20: ISorting pure virtual class definition.

3.2.4 Templates

A template is a programming feature that allows functions and classes to operate with generic types. Thanks to this feature, the function or class can work on many different data types without being rewritten for each one.

Vector

Some different templates where developed to let the framework and its core classes to manage different standard ANSI classes and to increase their functionality, as for example occurs with the `Vector` template class (code 3.21).

3.2.5 Namespaces

The namespaces are abstract containers created with the aim of holding a logical grouping of unique identifiers or symbols (ie. names of functions, variables, ...). An identifier defined

```
8   template <class object> class Vector : public std::vector<object>
9   {
10      public:
11          void insertAt(unsigned int iPos, object oItem)
12          {
13              if (iPos >= this->size())
14                  this->push_back(oItem);
15              else
16              {
17                  Vector<double>::iterator iter = this->begin() + iPos;
18                  this->insert(iter, oItem);
19              }
20          }
21  };
```

Code 3.21: Vector class template.

in a namespace is associated only with that namespace so, the same identifier can be independently defined in multiple namespaces; the meaning associated with an identifier defined in one namespace may or may not have the same meaning as the same identifier defined in another namespace. Therefore, namespaces are most often used to avoid naming collisions.

Relevant namespaces developed for the framework are:

`Configurator` namespace: namespace generated by the *protobuffers* library where the API and parsers for managing the configuration files is stored.

`Reconfiguration` namespace: main namespace where all the internal modules, classes, methods and other elements reside.

4 TECHNIQUES

In this chapter several techniques, related to the load balancing method and the domain decomposition procedure, used in the platform are presented. The aim of this chapter is not to present an actual state of the art, thing done on chapter 2, but to explain which techniques have been implemented in order to balance the workload among the nodes of the cluster and to decompose the global domain of the problem in several independent partitions.

The last section of this chapter introduces the reader of this document into the load imbalance, its consequences and measurement methods, although the platform does not perform any runtime optimisation action.

4.1 Load balancing

Before a problem can be executed on a parallel computing environment, the work to be done must be partitioned into domains among different processors. Due to uneven processor utilisation, load imbalance can cause poor performance.

Some indications regarding the total runtime must be taken into account [64]:

- The computation time required to complete a task, depends obviously direct on the complexity and size of the problem to solve but, other factors as the number of tasks used and the amount of available processing elements also affect. Using parallel machines in a distributed environment, the heterogeneity of the available machines affects directly on the computing time.
- The communication time depends not only on the location of the processors of the cluster, but also on the number of processes that runs the distributed application. Usually a differentiation between interprocessor and intra-processor communication is made: interprocessor communications are on behalf of the ones done between tasks running on different processing elements whereas the intra-processors communication answer

to the delays due to communications between tasks running on the same processing element. Communications of this last type are faster and it is even possible to take advantage of the characteristics of the architecture in multiprocessor computers, optimizing the global communication time. Moreover, the communication time between tasks running on different processing elements are generally higher due to the layers used for communication. In an idealized model of distributed memory parallel architecture, the cost of connecting two computers located at different tasks have two components: the time required for establishing the connection, which is a parameter of the communication network and the information transfer time, usually determined by the available bandwidth on the physical communication channel that connects the source with the destination computer.

- The time a task remains in a wait state is a consequence of the order in which operations are executed by a parallel application. It is related to the scheduler policies.

Although it is easy to find an algorithm to decompose a certain lattice among several processors into many parts of widely differing sizes, the difficult resides in predicting the size of each of those parts. After an initial distribution of work among the processors, some of them may run out of work much sooner than others and even faster in heterogeneous clusters. Hence, a dynamic balancing of nodes is needed in order to estimate the work to be assigned to each processor, balancing the idle time of each processor. Since none of the processors know how much work do they have, load balancing schemes which require this knowledge are not applicable.

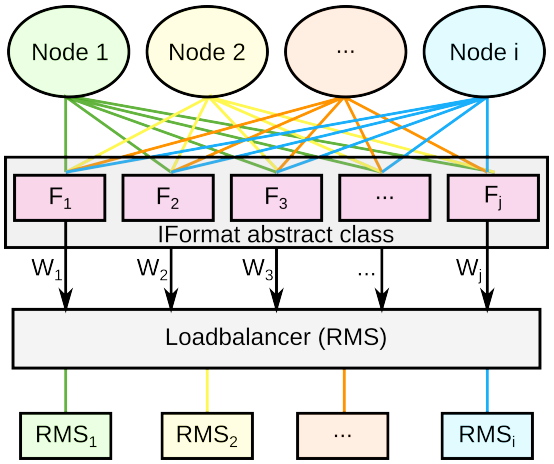


Figure 4.1: The RMS values are calculated as a result of a statistical function applied to the evaluation of several plugins over the characteristics of each machine of the system graph. Each loadable function F_j refers to the load balancing plugins implemented by the user (see page 56) in order to get the characteristics of the nodes of the system graph.

As it was explained before (see section 1.3) given a system graph G_s, the size of each generated domain, that corresponds with the granularity of the parallel solution, depends on the result of a statistical function, the RMS, applied to the characteristics of the nodes and the properties of the network that groups all these nodes. The RMS value for each processor is calculated by the following expression (see also figure 4.1):

$$\forall_{i \in P_{G_s}} RMS_i = \sum_j W_j \cdot F_{i,j} \tag{4.1}$$

where P_{G_s} is the set of vertices of G_s, W_j is the weight of each loadable function $F_{i,j}$ for a specific i node of the cluster.

It must be assured that, given an application graph G_a:

$$\sum_{i \in P_{G_s}} RMS_i = \|V_{G_a}\| \tag{4.2}$$

where $\|V_{G_a}\|$ is the total number of vertices of G_a that is, the factor to normalise the RMS_i values.

4.2 Domain decomposition

The fact of decomposing the application graph (see definition at page 50) into domains among processors could seem to be an easy task but, this is only true if the constraints over the neighbours of each vertex of the graph are not considered.

To measure how good or bad is a domain decomposition algorithm, a rate of failures can be used. This rate measures the relation between the number of needed elements that are not included in the partition and the number of elements that are part of that domain, that is, the number of elements that are dependencies of the elements of the partition and are not included in the partition over the total number of elements of the partition. This ratio depends on the pattern of communications (see figure 4.2).

As it has been explained before, the domain decomposition algorithm bases its actions on the information obtained from the load balancer shown in the previous section and on an arrangement of the vertices of the application graph. A vertex arrangement (or path) results from the application of an algorithm that, from an initial vertex of the graph, returns a sequence of all the vertices of the graph sorting them by a set of conditions. These conditions are set by the algorithm and the properties of the application graph (boundary conditions and neighbourhood).

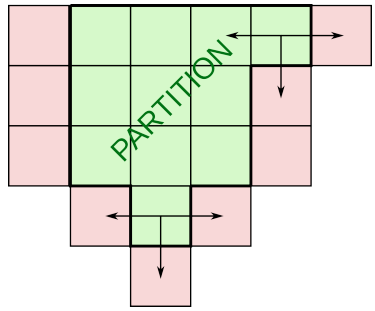


Figure 4.2: Rate of failures of a certain partition formed by the green squares. Red squares are the dependencies of the green ones that are not included in the partition. In this case, the dependencies for each vertex are defined by the Cartesian East, West and South directions. Given that partition and those directions, the rate of failure is $\frac{9}{11}$, that is, 9 elements must be added to the partition to assure that all the elements and dependencies are in the same partition.

Arrangement methods can be classified, whether the algorithm follows the links (edge) of each vertex, called local methods, or it generates an arrangement of the graph as a global entity without taking into account the dependencies of each of the vertices, that is non-local or global methods.

4.2.1 non-Local methods

These methods take advantages of the fact that neighbouring vertices are usually those that are closer to the actual one (minimal distance), that is, if $(x_n, x_{n-1}, \ldots, x_0)$ is the coordinates of a point $\vec{P}$ within the unit space and d is the distance along the curve when it reaches that point then, points that have nearby d values will also have nearby $(x_n, x_{n-1}, \ldots, x_0)$ values. Due to the properties of the memory architecture, accessing to the actual value of the vertex means the access to the same memory space of the neighbours.

Although these methods trend to generate bigger partitions (it depends on the symmetries of the problem), as they include elements in the partition that are not really dependencies of the vertices of the graph, they are faster than the local methods, where the neighbourhood area of each vertex is taken into account.

For the particular case of 2D problems, there are three typical domain decomposition methods that part the problem into, respectively, horizontal stripes, vertical stripes and a combination of both, the partition into square blocks:

Horizontal or vertical stripes: These methods (see figures 4.3 and 4.4) are easy to manage and to implement due to the continuous memory architecture of a computer and their execution are indeed fast but, they does not take into account the neighbourhood of the vertices. Due to this issue, they generate long border areas shared by each pair of neighbours. This implies a high probability of interaction between the two neighbour processors, increasing the rate of failures in the direction that does not fit with the orientation of the stripes. Using this decomposition algorithm the execution time spent in the simulation of a certain problem will not scale because of the great border areas in comparison with the total area of the sub-domain received by the slaves.

Square decomposition: Dividing the problem into squares (see figure 4.5) is a very similar technique to the stripes decomposition, but in this case, the sub-domain follows the geometry of the initial lattice. The ratio of communications between pairs of slave processors is negligible when the size of the lattice is much greater than zero ($d \gg 0$). The main drawback of this decomposition resides in the fact that the number of processes must be a square value and the resize of the domains is much difficult in comparison with the other methods.

Space-filling curves

Apart from these methods, as a non-local method the space filling curves [65–68] can be found. Space-filling curves are curves, whose range contains the entire 2D unit square or more generally, a N-dimensional hypercube. These curves give a mapping between 1D

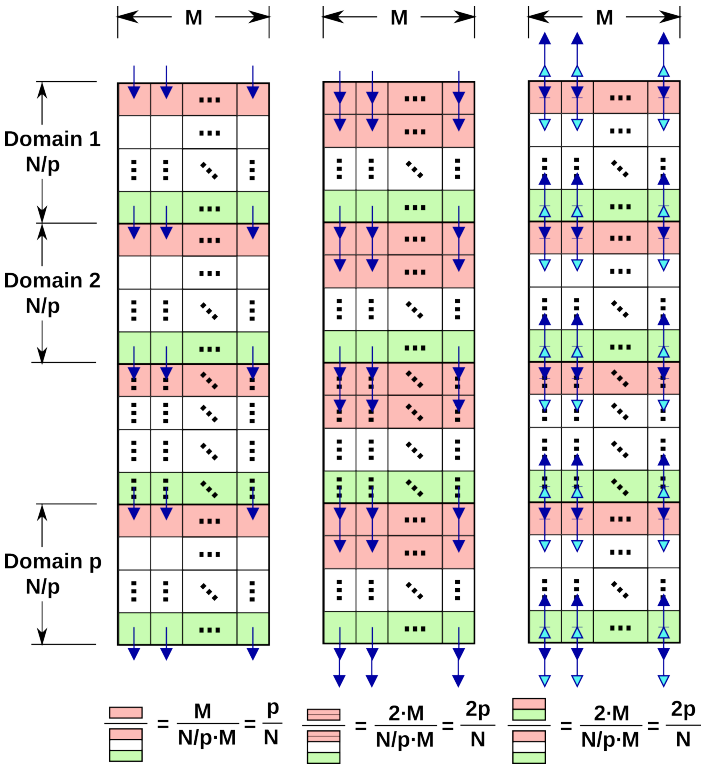


Figure 4.3: Possible domain decompositions on horizontal stripes and failure probability depending on the data dependencies of each problem cell.

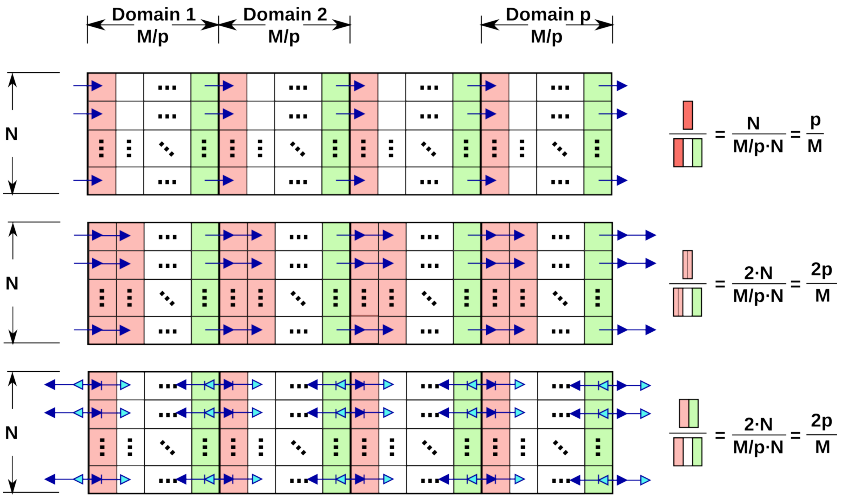


Figure 4.4: Possible domain decompositions on vertical stripes and failure probability depending on the data dependencies of each problem cell.

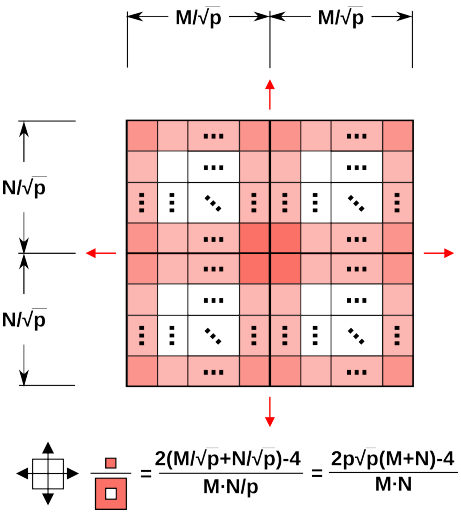


Figure 4.5: Possible square domain decompositions and failure probability.

and N-dimensional spaces that fairly well preserve locality. Space-filling curves have two characteristics that make them ideal for their use in parallelisation techniques [69, 70]:

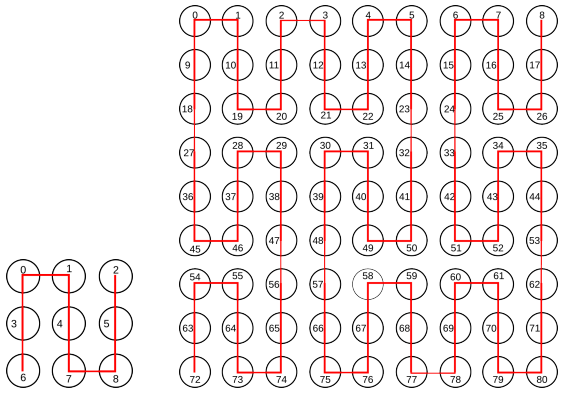
- They create a linear order of a multidimensional space.
- They preserve the neighbourhood of each vertex, that means, if two points are closely adjacent in the problem space (the one defined by the user algorithm), their index, given by the space filling curve, may differ only slightly. The set of points with close index can be summarized in the same partition.

Applying these two properties, it suggests that space-filling curves generate compact partitions [71]. As they are generated by a mathematical function that depends only on the dimensions of the lattice, these methods do not take into account the neighbours of the actual vertex, but they can generate the arrangement of vertices even faster than the three methods shown before.

The two decomposition methods based on space-filling curves provided by the platform are:

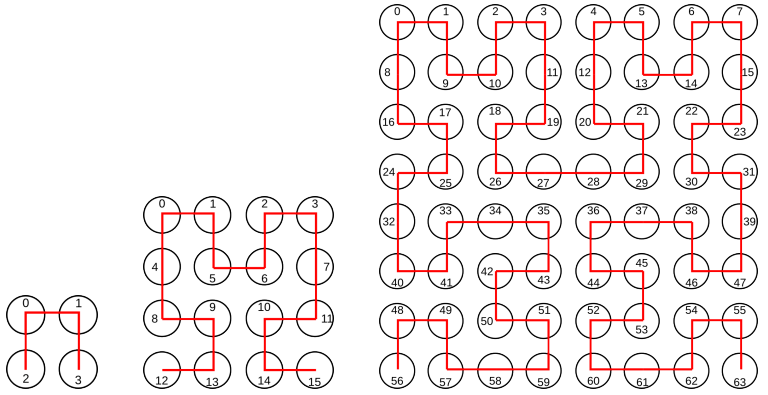
Peano curves: (see figure 4.6) due to its properties, it can be applied to lattices whose dimensions for 2D problems are $3^n \times 3^n$ elements.

Hilbert curves: (see figure 4.7) applicable to lattices whose dimensions in 2D are $2^n \times 2^n$. There are techniques to apply Hilbert curves to spaces with unequal side lengths [72, 73].



(a) First iteration. (b) Second iteration.

Figure 4.6: First two iterations of the Peano space-filling curve. The first iteration is defined over a lattice of $(D+1)^1 \times (D+1)^1$ elements and the second one over $(D+1)^2 \times (D+1)^2$ elements. For a general rule, the n-iteration of a Peano curve is defined over a lattice of $(D+1)^n \times (D+1)^n$ where $D = 2$, the number of dimensions of the lattice.



(a) First iteration. (b) Second iteration. (c) Third iteration.

Figure 4.7: First three iterations of the Hilbert space-filling curve. The first iteration is defined over a lattice of $D^1 \times D^1$ elements, the second one over one of $D^2 \times D^2$ elements and the third one over a lattice of $D^3 \times D^3$ elements. For a general rule, the n-iteration of a Hilbert curve is defined over a lattice of $D^n \times D^n$ where $D = 2$, the number of dimensions of the lattice.

Both curves can be easily generated by Lindenmayer systems, grammars where all non-terminal characters of a word must be replaced at the same step. A grammar can be defined as:

$$G = (T, nT, A, R) \tag{4.3}$$

where T stands for the set of terminal symbols of the words, nT defines the set of non-terminal characters, A specifies de axiom of the evolution (also called seed or initiator) and R are the production rules accepted by the grammar to recursively evolve the system. The assignment/substitution operator is identified by the symbol ":=".

In the particular case of the Peano curves, these parameters are defined as (see figure 4.8):

$$\begin{aligned} T &= \{\uparrow, \rightarrow, \downarrow, \leftarrow\} \\ nT &= \{p, q, b, d\} \\ A &= \{p\} \\ R &= \begin{cases} p := p \uparrow q \uparrow p \rightarrow b \downarrow d \downarrow b \rightarrow p \uparrow q \uparrow p \\ q := q \uparrow p \uparrow q \leftarrow d \downarrow b \downarrow d \leftarrow q \uparrow p \uparrow q \\ b := b \downarrow d \downarrow b \rightarrow p \uparrow q \uparrow p \rightarrow b \downarrow d \downarrow b \\ d := d \downarrow b \downarrow d \leftarrow q \uparrow p \uparrow q \leftarrow d \downarrow b \downarrow d \end{cases} \end{aligned} \tag{4.4}$$

where $\uparrow$, $\rightarrow$, $\downarrow$ and $\leftarrow$ specifies the movement that must be done to draw the union between to consecutive components of the curve "without lifting the pencil".

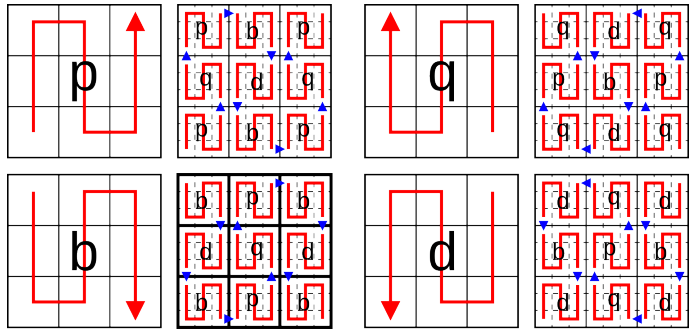


Figure 4.8: Construction of the Peano space-filling curve from its grammar.

The problem of the Peano space-filling curve is that it generates long links between some vertices, as for example the link between vertex 5 and vertex 50 or vertex 30 to vertex 75, at figure 4.6(b). Furthermore, it cannot always assure a compact domain decomposition.

On the other hand, to build a Hilbert curve, the following parameters might be used (see figure 4.9):

$$
\begin{aligned}
T &= \{\uparrow, \rightarrow, \downarrow, \leftarrow\} \\
nT &= \{p, q, b, d\} \\
A &= \{p\} \\
R &= \begin{cases} p := q \uparrow p \rightarrow p \downarrow b \\ q := p \rightarrow p \uparrow q \leftarrow d \\ d := b \downarrow d \leftarrow d \uparrow q \\ b := d \leftarrow b \downarrow b \rightarrow p \end{cases}
\end{aligned}
\tag{4.5}
$$

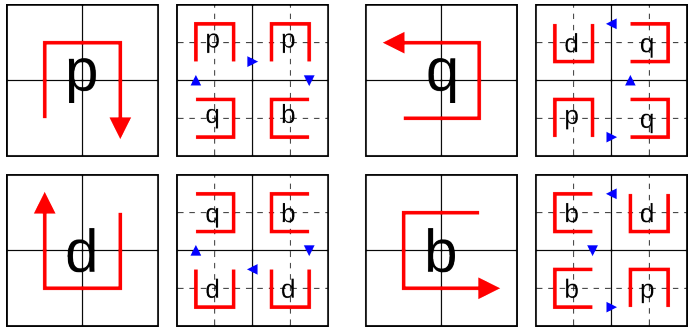


Figure 4.9: Construction of the Hilbert space-filling curve from its grammar.

Iterating over the equation 4.5, a mathematical recursive expression for each dimension of the problem can be obtained:

$$
\begin{aligned}
H_x^n &= \frac{1}{2}\left[\frac{-1}{2} + H_y^{n-1}, \frac{-1}{2} + H_x^{n-1}, \frac{1}{2} + H_x^{n-1}, \frac{1}{2} - H_y^{n-1}\right] \\
H_y^n &= \frac{1}{2}\left[\frac{-1}{2} + H_x^{n-1}, \frac{1}{2} + H_y^{n-1}, \frac{1}{2} + H_y^{n-1}, \frac{-1}{2} - H_x^{n-1}\right]
\end{aligned}
\tag{4.6}
$$

where $H_x^0 = H_y^0 = 0$, base case of the recursion. Between these both equations, the target element in the $[0,1] \times [0,1]$ Cartesian set is returned. To transform this position into an element of the interval $[0, N]$ (being N the number of elements of the graph), the next

expressions must be applied:

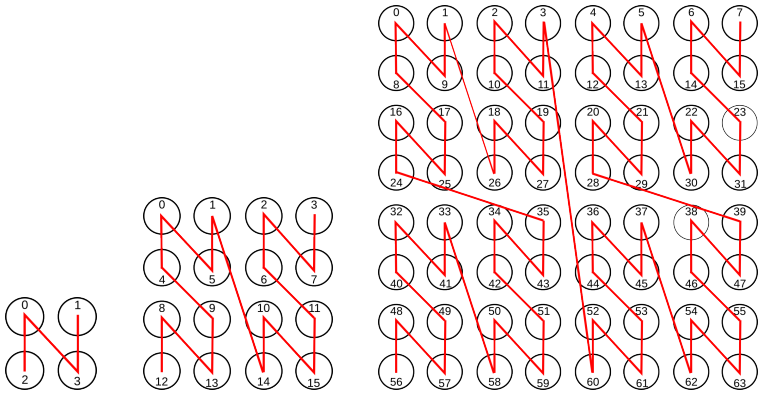
$$\begin{aligned} x &= 2^n H_x^n + 2^{n-1} - \frac{1}{2} \\ y &= 2^n H_y^n + 2^{n-1} - \frac{1}{2} \\ z &= x + 2^n y \end{aligned} \tag{4.7}$$

where x and y are defined in the $[0, 1]$ space interval, z returns values in the interval $[0, N]$ and n stands for the number of iteration that must be executed to reach the N value:

$$n = \log_2 \frac{N}{2} \tag{4.8}$$

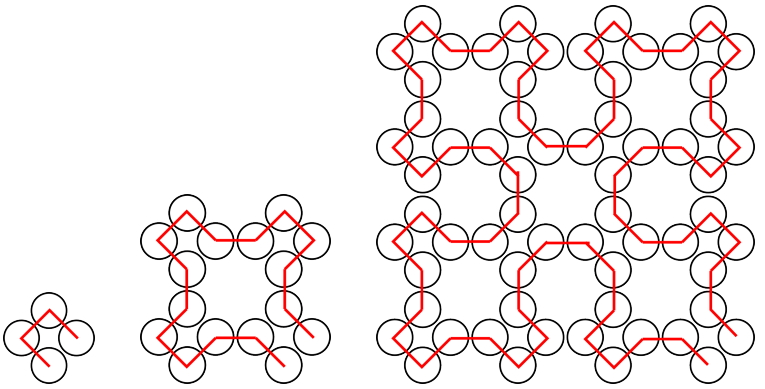
Making a comparison between the Peano space-filling curve and the Hilbert one [74], regardless of the dimension of the problem, the partitions made applying the second one are more compact that the first one, because the links between each pair of neighbour vertices are shorter.

Other space-filling curves, as *Lebesgue* (refer to figure 4.10) or *Sierpinski* (see figure 4.11) curves, are typically used but they are not provided by default by the platform.



(a) First iteration. (b) Second iteration. (c) Third iteration.

Figure 4.10: First three iterations of the Lebesgue space-filling curve. The first iteration is defined over a lattice of $D^1 \times D^1$ elements, the second one over one of $D^2 \times D^2$ elements and the third one over a lattice of $D^3 \times D^3$ elements. For a general rule, the n-iteration of a Lebesgue curve is defined over a lattice of $D^n \times D^n$ where $D = 2$, the number of dimensions of the lattice.



(a) First iteration. (b) Second iteration. (c) Third iteration.

Figure 4.11: First three iterations of the Sierpinski space-filling curve. This curve is a Hilberts curve applied to a triangular-based lattice.

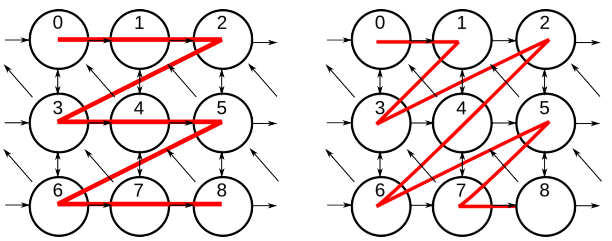
4.2.2 Local methods

On the other hand, there are other methods that are sightly more complex to implement but they take into account the neighbouring of each vertex of the graph, being this characteristic an important feature to consider when the set of vertices have heterogeneous neighbourhoods. This methods are generally implemented over a stack, being the stack policy (FCFS or LCFS) the one that determines the behaviour of the local method.

The two standard local methods included and provided by the platform are:

DFS (Depth First Search): expands all the vertices of a link before backtracking to consider another link.

BFS (Breadth First Search): expands all vertices before getting into a deeper level.



(a) Depth First Search Algorithm (b) Breath First Search Algorithm

Figure 4.12: Local domain decomposition methods

DFS

It progresses by expanding the first neighbour vertex of set of visited vertices, going deeper and deeper until there are no more child nodes on that branch, backtracking then to the nearest father node and starting again the search (see figure 4.12(a)).

Using a LCFS stack, the algorithm can proceeds as shown in figure 4.1.

```
name : DFS
input : Graph
output: Arrangement in which the vertices have been visited
/* forests                                                    */
while nonVisitedVertex(Graph) do
    root ← firstNonVisitedVertex()
    Stack.Push (root)
    root.visited ← true
    while Stack.Empty()=false do
        vertex ← Stack.Pop()
        Arrangement [i] ← vertex.index
        foreach neighbour ∈ vertex do
            Stack.Push (neighbour)
            neighbour.visited ← true
        end
    end
end
```

Algorithm 4.1: Pseudocode of the DFS algorithm using a LCFS stack. The stack structure inserts elements on and takes elements from the head of it.

BFS

It progresses by exploring all the neighbours vertices of a visited vertex. For each of those nearest nodes, it explores their unexplored neighbours until visiting all the nodes of the graph. There is no backtracking (see figure 4.12(b)).

Using a FCFS stack (queue), the algorithm can proceeds as shown in figure 4.2.

4.2.3 Comparative between basic domain decomposition methods

In the next table (see table 4.1) the different domain decomposition methods shown before are compared, as a sum up.

```
name  : BFS
input  : Graph
output: Arrangement in which the vertices have been visited
/* forests                                                    */
while nonVisitedVertex(Graph) do
    root ← firstNonVisitedVertex()
    Queue.Push (root)
    root.visited ← true
    while Queue.Empty()=false do
        vertex ← Queue.Pop()
        Arrangement [i] ← vertex.index
        foreach neighbour ∈ vertex do
            Queue.Push (neighbour)
            neighbour.visited ← true
        end
    end
end
```

Algorithm 4.2: Pseudocode of the BFS algorithm using a FCFS stack (queue). The queue structure inserts elements on the head of the structure and takes elements of the tail.

Method	Complexity	Locality	Applicable if	Advantage	Disadvantage
Stripes	$O(N)$	non-Local	Links in the direction of the stripe	Fast. Memory continuity property	Long borders
Squares	$O(N)$	non-Local	Size per dimension power of two	Similar to stripes, mixing directions	non-resizeable
Peano	$O(\log_3 \frac{N}{3})$	non-Local	Size per dimension power of three	Fast and easy to resize the domains	Long borders
Hilbert	$O(\log_2 \frac{N}{2})$	non-Local	Size per dimension power of two	Fast and easy to resize the domains	–
DFS	$O(N+M)$	Local	Heterogeneous communication pattern	Domain fits with the topology	Stack based
BFS	$O(N+M)$	Local	Heterogeneous communication pattern	Domain fits with the topology	Stack based

Table 4.1: Comparison table between the domain decomposition methods shown above. N stands for the number of vertices of the graph whereas M stands for the total number of links of the graph.

4.2.4 Penalty of choosing an inappropriate domain decomposition method

Choosing an inappropriate domain decomposition algorithm entails different penalties from different points of view:

- As more compact is the partition, as less RO values must be included in the partition and the partitions are therefore smaller. It is worth to remember that the difference between a partition and a domain lies on the fact that the partition is the serialisation of a domain. The ideal partition, where the number of extra vertices added (RO values) is 0, does not exist unless we are considering forests of graphs with only one vertex. A compact partition implies that the dependencies of the vertices are also RW on the same partition, so that the set of RO values is mainly subsumed in the RW set. The partition size grows, in this case, with the number of extra vertices that are added to the partition, but not with the number of links, since the dependencies of the RO elements are removed from the partition before it is encapsulated.
- As bigger is the partition, as bigger is the buffer needed to encapsulate the partition and hence, the datagrams sent by the master to the slaves with the information of the partition are bigger, implying a more time to send the buffer to the slave nodes. This can affect the load imbalance of the application by growing it.
- Depending on the algorithm chosen, the time needed to create the partitions varies and therefore increasing the load imbalance of the application.

To evaluate how precise is a partitioning algorithm, a metric somehow based on the number of extra elements per partition can be used (see figures 4.13, 4.14 and 4.15). The metric should therefore strength the fact that it does not add extra vertices to partitions, that is, promote compact partitions. Any certain partition P_i with N_{P_i} vertices has $N^w_{P_i}$ RW elements (effective nodes) and $N^r_{P_i}$ RO elements as dependencies (non-effective nodes), in the way that $N^w_{P_i} + N^r_{P_i} = N_{P_i}$. The other factors involved in the partition algorithm are the number of domains, D, into which the application graph has been parted and the cardinality of the application graph.

As a first try, the effectiveness Eff of the algorithm can be measured by an arithmetic mean (average) (see equation 4.9):

$$Eff_{am}(\%) = 100\left(1 - \frac{\sum_{P_i \in G_a} N^r_{P_i}}{|G_a|}\right) \tag{4.9}$$

where Eff_{am} is the arithmetic mean effectiveness of a partition algorithm, P_i refers to the partition i, G_a is the application graph with cardinality $|G_a|$ and $N^r_{P_i}$ stands for the number of non-effective vertices of the partition P_i.

In the ideal scenario where there are no dependencies between the vertices of the graph (all vertices are effective nodes), the effectiveness of the arithmetic mean is $Eff_{am} = 100\%$ but, the problem of using a metric based on an arithmetic mean is that it does not show a clearly advance between situations in which a certain partition has not non-effective nodes. They way to emphasize or distinguish this is by using a geometric mean (see equation 4.10) but not basing it on the $N_{P_i^r}$ elements but on the granularity of the partition P_i:

$$Eff_{gm}(\%) = 100 \left(1 - \frac{\sqrt[D]{\prod_{P_i \in G_a} N_{P_i}}}{|G_a|} \right) \tag{4.10}$$

where Eff_{gm} is the geometric mean effectiveness of a partition algorithm, D is the number of domains, P_i is each partition of the application graph with cardinality $|G_a|$.

In the ideal situation where there are partitions with only effective vertices, the effectiveness of the geometric mean is $Eff_{gm} = 100\%$.

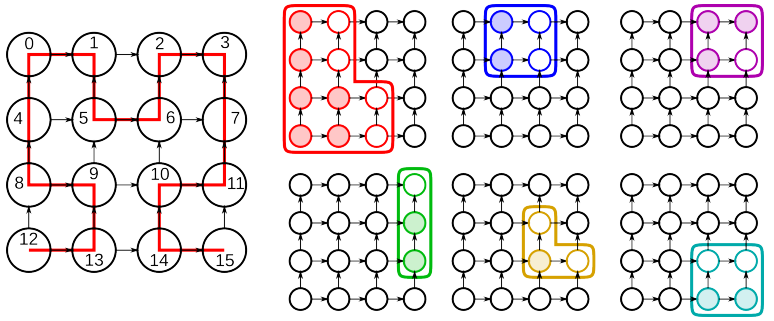


Figure 4.13: Effectiveness of the Hilbert domain decomposition method. In this case, the application graph G_a is composed by 16 vertices, each of them with North and East dependencies and decomposed into 6 domains with RMS values of $\{6, 2, 3, 2, 1, 2\}$. The coloured circles identify the RW vertices, whereas the white circles bordered by a coloured line identify the RO nodes and the continuous red line follows the arrangement generated by the sorting algorithm. When serializing the partitions, the neighbours of the RO elements are deleted but, they have been kept for this example in order to identify the corresponding areas of each partitions regarding the global application graph. $Eff_{am} = \left(1 - \frac{4+2+1+1+2+2}{16}\right) * 100 = 25\%$, $Eff_{gm} = 100 \left(1 - \frac{\sqrt[6]{10 \times 4 \times 4 \times 3 \times 3 \times 4}}{16}\right) = 73.54\%$

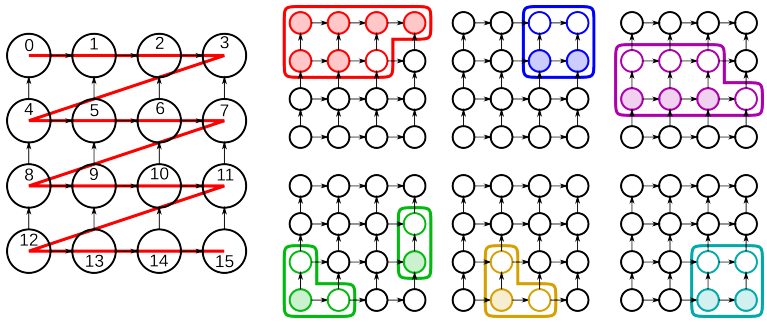


Figure 4.14: Effectiveness of the Horizontal stripes domain decomposition method. In this case, the application graph G_a is composed by 16 vertices, each of them with North and East dependencies and decomposed into 6 domains with RMS values of $\{6, 2, 3, 2, 1, 2\}$. The coloured circles identify the RW vertices, whereas the white circles bordered by a coloured line identify the RO nodes and the continuous red line follows the arrangement generated by the sorting algorithm. When serializing the partitions, the neighbours of the RO elements are deleted but, they have been kept for this example in order to identify the corresponding areas of each partitions regarding the global application graph. $Eff_{am} = 100\left(1 - \frac{1+2+4+3+2+2}{16}\right) = 12.5\%$, $Eff_{gm} = 100\left(1 - \frac{\sqrt[6]{7\times4\times7\times5\times3\times4}}{16}\right) = 70.19\%$

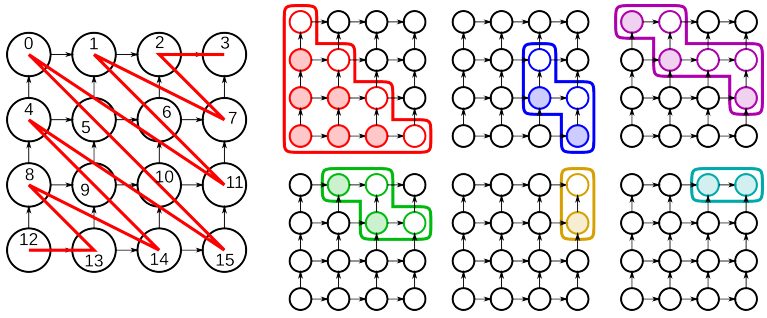


Figure 4.15: Effectiveness of the BFS domain decomposition method. In this case, the application graph G_a is composed by 16 vertices, each of them with North and East dependencies and decomposed into 6 domains with RMS values of $\{6, 2, 3, 2, 1, 2\}$. The coloured circles identify the RW vertices, whereas the white circles bordered by a coloured line identify the RO nodes and the continuous red line follows the arrangement generated by the sorting algorithm. When serializing the partitions, the neighbours of the RO elements are deleted but, they have been kept for this example in order to identify the corresponding areas of each partitions regarding the global application graph. $Eff_{am} = 100\left(1 - \frac{4+2+3+2+1+0}{16}\right) = 12.5\%$, $Eff_{gm} = 100\left(1 - \frac{\sqrt[6]{10*4*6*4*2*2}}{16}\right) = 75.27\%$. As the arithmetic mean of the 4.13 and the 4.15 have the same value, the geometric mean allows us to advice a better performance of the BFS algorithm.

4.3 Performance measurement

Performance tools are software systems that assist programmers in understanding the runtime behaviour of their application on real systems and ultimately in optimizing the application with respect to execution time, scalability, or resource utilization.

For the measurement of the performance of the framework, the *Scalasca* tool set has been used. *Scalasca* [75] is a performance tool set that has been specifically designed to analyse parallel application execution behaviour on large-scale systems. It offers an incremental performance-analysis procedure that integrates runtime summaries with studies of concurrent behaviour via event tracing.

Measuring and application implies the instrumentation of the user application programs, i.e., insertion of special measurement calls at specific important points (events) points during the application run. The tool first generates instrumentation code that serves as entry points for performance data collection (*instrumentation* phase). Next, the application and instrumentation code are executed on the target platform and raw performance data are collected at runtime by the tool (*measurement* phase). The tool organizes the raw data and can optionally perform various automatic analyses to discover and perhaps suggest resolutions to performance bottlenecks (*analysis* phase). When tracing is enabled, each process generates a trace file containing records for all its process-local events. After program termination, the profiler reloads the trace files back into main memory and analyses them in parallel using as many CPU units as have been used for the target application itself. During the analysis, the profiler searches for characteristic patterns indicating wait states and related performance properties, classifies detected instances by category and quantifies their significance. The result is a pattern-analysis report similar in structure to a summary report but enriched with higher-level communication and synchronization inefficiency metrics. Both summary and pattern reports contain performance metrics for every function call path and system resource.

As an alternative to *Scalasca*, some other trace browser tools such as *Paraver* [76], *Periscope* [77] or *Vampir* [78] can be used.

A way of measuring how well or bad has an application been parallelised, is by measuring the load imbalance. Load imbalance can be measured regarding two concepts: “imbalance percentage” and “imbalance time”. Imbalance percentage provides an idea of the “badness” of the load imbalance in the range of 0 to 100, where a perfectly balanced code segment would have have an imbalance of 0% and a serial portion of a code segment on a parallel application (for example a serial I/O) would have imbalance percentage of 100. The load imbalance percentage corresponds to the percentage of time that the rest of the processes, excluding the slowest processing element, is not engaged in useful work on the

given function. However, a section of code that has a high imbalance percentage should not necessarily be the main target of performance optimization. In the example of above, a serial I/O will always have load imbalance percentage of 100, independently of the amount of time that it takes. If the fraction of time spent on that particular I/O operation is small, its impact on the overall performance of the application will be negligible. Thus, a metric related to the execution time is needed, in order to identify regions of the program that should be considered for optimization, the imbalanced time.

There are many situations regarding the communication and synchronisation process between parallel processes that can spoil the performance of a parallel application program. This situations and how they affect to the efficiency of the application can be measured using MPI-related metrics:

- communication patterns.
- synchronisation patterns.

4.3.1 Communication related metrics

Several communication patterns can be found, according to the moment in which the sender/receiver process sends/receives the information:

- Collective communications.
- P2P communications.

Collective communications

The collective metrics related to communications can be subdivided into:

Early reduce: Collective communication operations that send data from all processes to one destination process may suffer from waiting times if the destination process enters the operation before any data could have been sent. The pattern refers to the time lost as a result of this situation (see figure 4.16). It applies to MPI calls `MPI_Reduce()`, `MPI_Gather()` and `MPI_Gatherv()` (See page 239 for more information).

Early scan: Waiting time due to the situation in which a process enters a reduction operation earlier than its sending counterparts (see figure 4.17). It applies to `MPI_Scan()`.

Late broadcast: Operations that send data from one source process to all processes may suffer it if destination processes enter the operation earlier than the source process, that is, before any data could have been sent (see figure 4.18). It applies to `MPI_Bcast()`, `MPI_Scatter()` and `MPI_Scatterv()` (See page 239 for more

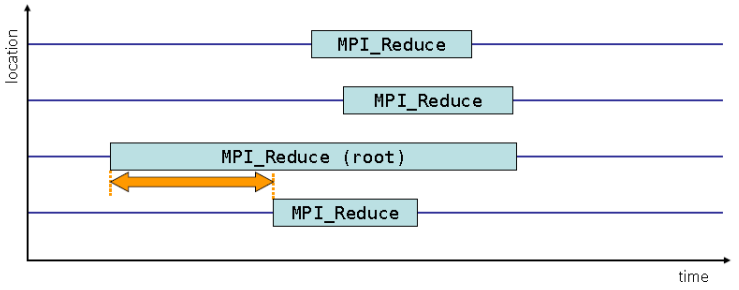


Figure 4.16: Early reduce: the destination process enters the operation before any data could have been sent.

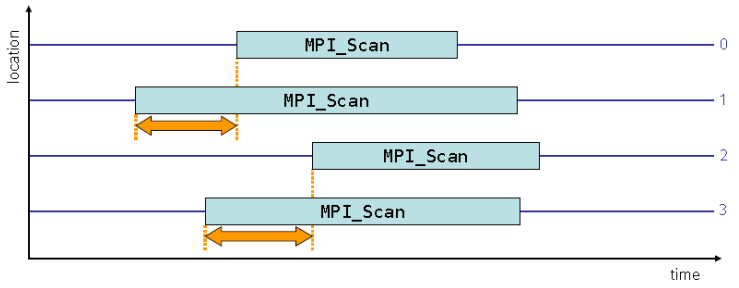


Figure 4.17: Early scan: a process enters a reduction operation before its partners sent the data.

information).

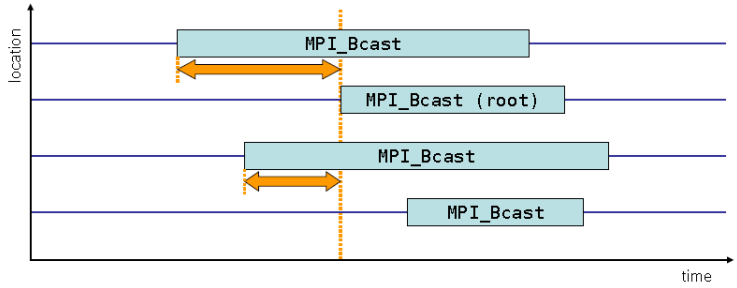


Figure 4.18: Late broadcast: a process sends data in broadcast mode after the destination processes enters the operation.

Wait at $N \times N$: Operations that send data from all processes to all processes and can not finish the operation until the last process has started it (see image 4.19). This pattern covers the time spent in n-to-n operations until all processes have reached it. It applies to `MPI_Reduce_scatter()`, `MPI_Allgather()`, `MPI_Allgatherv()`, `MPI_Allreduce()`, `MPI_Alltoall()` and `MPI_Alltoallv()` (See page 239 for more information).

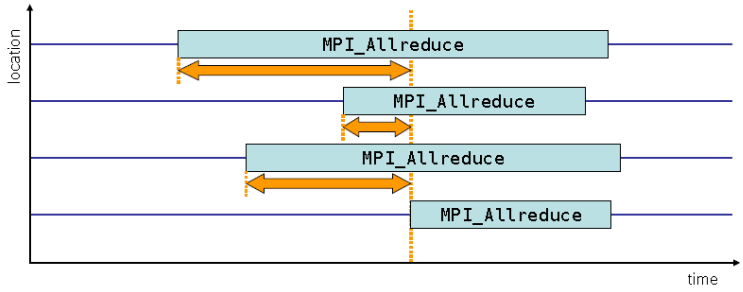


Figure 4.19: Wait at $N \times N$: Sent of data from all processes to all processes and can not finish before the last process has started it.

$N \times N$ completion: Time spent in synchronizing collective operations after the first process has left the operation (see image 4.20). It applies to: `MPI_Allreduce()`, `MPI_Alltoall()`, `MPI_Alltoallv()`, `MPI_Allgather()`, `MPI_Allgatherv()` and `MPI_Reduce_scatter()` (See page 239 for more information).

P2P communications

The peer-to-peer communications can be measured by the following metrics:

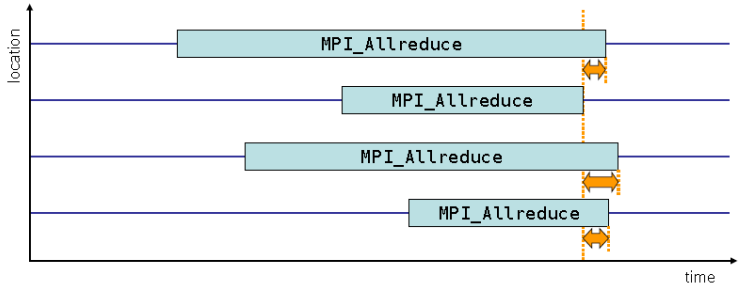


Figure 4.20: $N \times N$ Completion: Synchronisation after the first process lefts the operation.

Late receiver: A send operation is blocked until the corresponding receive operation is called (see figure 4.21), either because the MPI implementation is working in synchronous mode by default or either cause the size of the message to be sent exceeds the available MPI-internal buffer space and the operation is blocked until the data is transferred to the receiver. Applies to blocking send operations (See page 239 for more information).

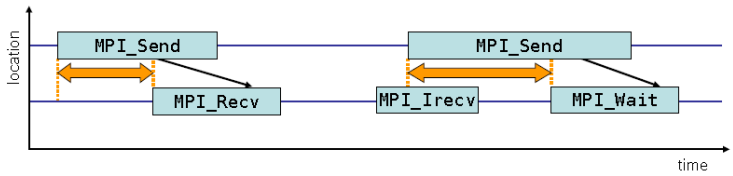


Figure 4.21: Late receiver: Block of a send operation until the receive operation is called.

Late sender: Caused by a blocking receive operation such as `MPI_Recv()` or `MPI_Wait()` posted earlier than the corresponding send operation (see figure 4.22).

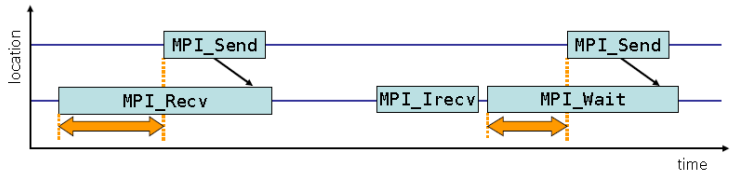


Figure 4.22: Late sender: Block of a receive operation that starts before the corresponding send operation.

4.3.2 Synchronisation related metrics

As synchronisation metrics, the next subcategories can be found depending on the instant in which the different processors enter or leave the corresponding function:

Collective barriers

Wait at barrier: Covers the time spent waiting in front of a MPI barrier, which is the time inside the barrier call until the last processes has reached the barrier (see figure 4.23).

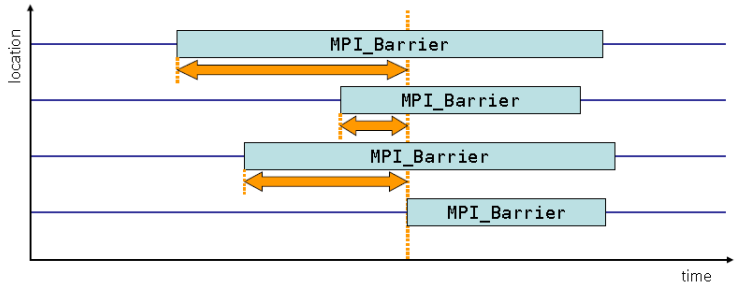


Figure 4.23: Wait at barrier: Time spent waiting inside a barrier.

Barrier completion: This pattern refers to the time spent in MPI barriers after the first process has left the operation (see figure 4.24).

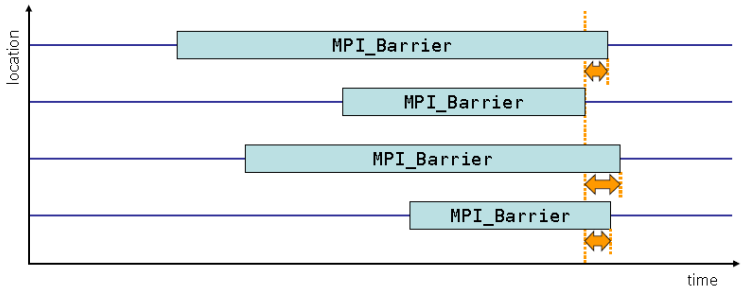


Figure 4.24: Barrier completion: Extra time spent waiting inside a barrier after the first process left it.

5 Using the framework

Notwithstanding of being able to use the proposed framework, an example of the complete design and implementation of a test application must be explained but, in order to complete this example, the most basic concepts regarding the framework and how it works may be understood.

In the next sections, a brief explanation on how the platform works will be introduced, following with the clarifications of the implementation of a simple test application that can be used to consolidate the previous acquired ideas and concepts. But, the implementation of the application is not enough to run it parallel and it is needed to configure it. This chapter ends with the description of the filesystem and directories hierarchy that the platform need, in order to run the user applications integrated with the framework.

5.1 How does the framework work

How the framework works depends basically on how the master (really supermaster) process works and which behaviour has been defined for the slaves processes. This behaviour is chosen by implementing a method called by a callback method and setting the corresponding parameters in diverse configuration files.

5.1.1 Master execution

The implementation of the master process can be summarized in four basic steps, without meaning this, that this process performs only these tasks. The execution of the master process, regardless of the communication with other slave processes, starts by creating the system graph (see expression 1.1 and page 50) and application graph (see expression 1.3 and page 50). As the system graph involves the reception of messages from the slaves

that have recently joined the simulation, the explanation of how this graph is created is now omitted and it will be explained in section 5.5. Once this action is performed, the graph has been instanced and set with the initial values and the slaves have join the simulation, the load balancing procedure can start, leading in, upon completion, to the domain decomposition procedure. At the end of each step of the simulation, once the master process has received the results of all the slaves, it gathers them to update the graph structure.

Graph creation

The algorithm to create the graph from the multidimensional matrix defined by the user in the correspondent configuration file (see page 106), is one of the most complex parts of the execution of the master process and it is hardly difficult to being optimized but, at least, it is executed only once per master execution, that is, the application graph does not change during a complete simulation of an user algorithm.

Given a multidimensional matrix, a set of directions which define the data dependencies between the values of that matrix (topology) and the boundary conditions of the it, the resulting graph is completely unique (see figure 5.1). The complexity (runtime big-O) of the algorithm to create the graph is $O(N^2 \cdot M)$ where N is the length of the vertices array of the graph and M is the length of the array of the directions defined by the user.

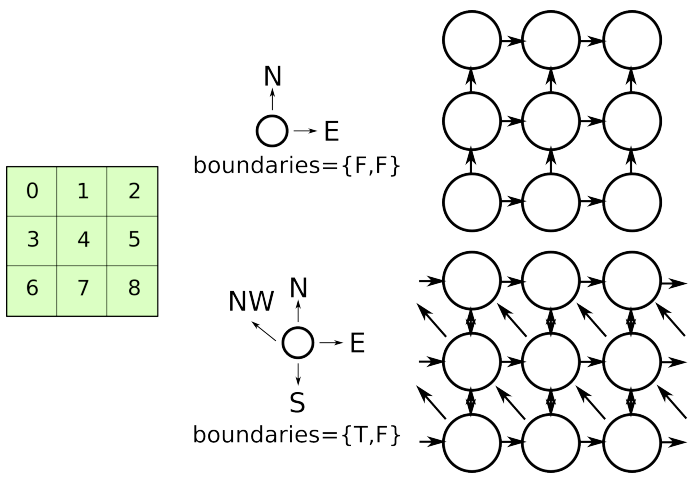


Figure 5.1: Graph representation of the application graph. Direction offsets depend on problem dimensions, the topology and the boundaries.

Load balancing procedure

The execution of the load balancing procedure is carried out by the master process before the domain decomposition procedure is called.

The load balancing algorithm returns the number of elements per domain into which the application graph must be parted. This calculus is related to a normalized value as a result of a weighted combination of the measured features of the nodes of the cluster (see expression 4.1). The user can use each of these features by choosing the corresponding plugin from the standard plugins set or even implementing his own ones (refer to sections 3.2.3 and 5.3.2 to know how the load balancing plugins are implemented and selected).

The runtime of this algorithm is $O(N)$ because a RMS value has to be calculated for all nodes of the cluster, N.

Domain decomposition procedure

The execution of the partitioner algorithm is carried out by the master process once a certain number of steps have been executed or a certain condition is fulfilled. This certain number of steps is given by the user and specified in the partitioner configuration file (see page 104) at the `Heuristic` structure section of the configuration file.

The domain decomposer algorithm has as input values the amount of domains to create and the normalized number of tasks per partition.

This procedure sums up four internal actions:

1.– As it has been explained before, the domain decomposition algorithm firstly generates an arrangement of vertices of, in this case, the application graph (see figure 5.2). The algorithm to generate the sort of vertices can be implemented by the user in a plugin or selected from the standard set of sorting plugins (refer to sections 3.2.3 and 5.3.3 to get information about how the sorting plugins are implemented and selected). It is also possible to use more than one plugin for generate the arrangement of vertices. The use of multiple plugins to generate the arrangement of vertices makes sense since there are problems that, depending on the step of the simulation, decomposes the problem in differently.

2.– Once the sorting of vertices has been calculated, the partition can be determined by joining as many elements of the sorting array as the RMS value that has been calculated for each of the nodes of the system graph (see figure 5.3). The values that correspond to a certain node are marked as RW values and the frontiers of these ones, as RO values. The union of both kind of nodes defines the partition

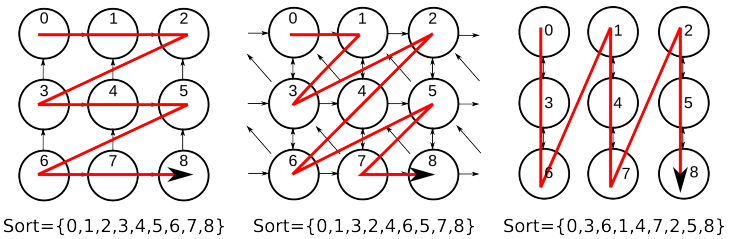


Figure 5.2: Arrangement of vertices of the application graph. The arrangement depends on the sorting plugin loaded. In this example, the plugin that generates that sorting array is the BFS algorithm.

that must be sent to a certain node. It is important to highlight that a partition is not more than a subgraph sent to a certain node that has as many RW elements as the RMS value of that node.

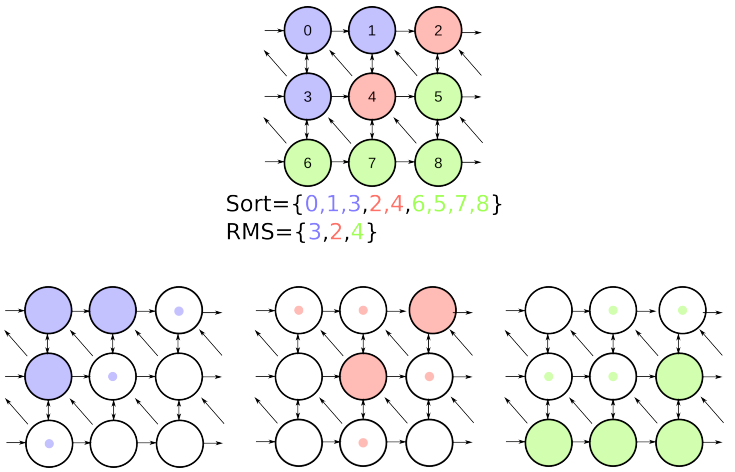


Figure 5.3: Partition of the application graph regarding the values returned by the RMS algorithm. The first node receives three elements (0,1,3); the second one only two, (2,4); and the third one, the last four values (6,5,7,8). The little circle inside the vertices of the graph represents those vertices with RO status.

3.– As the RO vertices only provide data values to the RW ones, they are not updated so, all the dependencies of those nodes can be safely deleted from the partition (see figure 5.4). In this way, the partition graph has been optimized, reducing the size of the network datagrams as they are sent through the network.

4.– The last step involves the encapsulation of the partition into a stream buffer following the recommendations made by MPI-Forum, as described in figure 3.4.

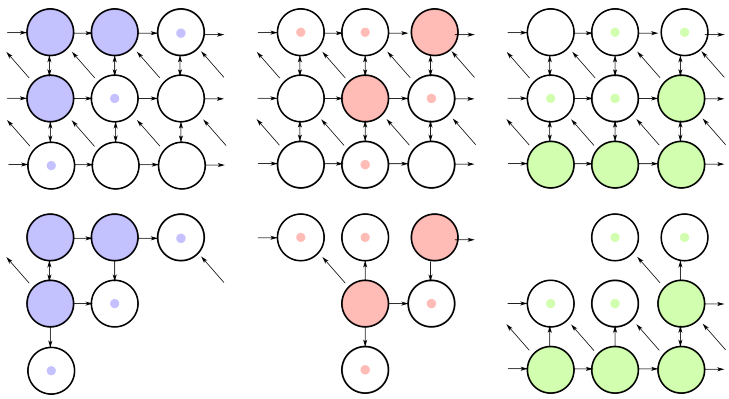


Figure 5.4: Partition optimization. Nodes with a point inside represent the RO vertices of the partition. As the RO nodes are not updated (they appear as RW in other partition), they do not have dependencies and thus, they do not need output connections with other nodes. Nodes that are not part of the RO or RW sets, can be eliminated from the partition.

Gathering the results

As soon as all the slave processes send its results to the master process, this process must update the application graph with the new values calculated by the slaves. The slaves send the partition with the updated values back to the master process. This partition has been encapsulated by the slave process in the same way as the master do at the beginning of the step.

As the RO values have not been updated, the only thing that implies gathering the results of the slave consists on restoring the indices of the RW values of the partition buffer and copying these ones into the corresponding vertex of the application graph.

5.1.2 Slaves execution

The behaviour of the slaves processes is more like the one specified by the user in the configuration files than in the case of the master process. Without taking into account the communication procedure of each slave process with its master, the behaviour of the slave process is completely defined on a special method, `kernel`, implemented by the user and called by callback from the slave-side of the framework (see platform architecture on page 32 and kernel implementation on page 96).

The communication part of the execution of the slaves works as an skeleton where, depending on the tag received from the master process, the slave executes and calls certain methods to attend the actual request.

5.1.3 Timeline

At the moment, only the computation procedures of both kind of processes have been explained and the specification of the network datagrams and synchronisation procedure of the processes have been omitted from the previous section.

Once it is understood how the platform works, the timeline of the application, from the point of view of the messages that the slaves processes exchange with the master, can be introduced. The timeline can be divided into three phases, sorted in chronological order:

Handshake: The opening phase (see image 5.5) is initiated by the slave process (MPI `TAG_JOIN` message) and confirmed by the master process (MPI `TAG_CONFIGURE` message). As the master needs to know the characteristics of the slave process as a node of the cluster (that is, the CPU frequency and the amount of free memory) and the slave can not access to the cluster specifications configuration file, the slave has to send its hostname to the master, allowing this one to search the node on the corresponding configuration file to get the needed values. The request sent by the slave is answered by the master sending a message with the coordinates specified by the user in the configuration file that defines the problem and with the dimensions of the problem because, as it has been explained before, the offset directions depends on the boundaries, the topology and the dimensions of the problem (refer to explanation on page 84). The MPI `TAG_JOIN` message sizes 255 Bytes whereas the size of the `TAG_CONFIGURE` message depends on the values configured by the user, as the problem topology, dimensions of the problem and number of nodes of the partition.
Once all the slaves have joined the execution, the second phase can start.

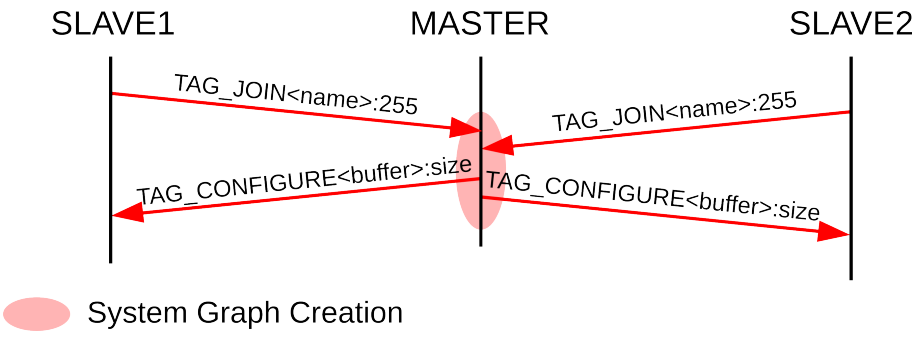


Figure 5.5: Handshake phase between two slaves with a master process. The `TAG_JOIN` request it sent by the process slave and answered by the master with a `TAG_CONFIGURE` message

Execution phase: The execution or second phase covers mainly the execution of the algorithm by the slaves processes and the gathering of results by the master process (see

figure 5.6), from the first TAG_PARTITION MPI message until the reception of the tag TAG_FINALIZE in the master side. If in the actual step the heuristic function (see page 103) determines that on this step a the RMS coefficients must be adjusted, the master executes the load balancing and domain decomposition methods based on the loaded plugins and sends to the active slaves the MPI TAG_PARTITION, with the new partitions, and the TAG_DATA, with the values of the vertices of the graph, messages; if the load balancing schema must not be updated, the partition remains constant so, in this case, the only parameter that has changed and that must be sent is the values of the vertices of the graph. This values are sent under the tag identifier TAG_DATA and its size depends on the datatype specified by the user and the amount of values, that is, the RMS value calculated for that slave process.

From the point of view of the slave process, the results are sent to the master process using a TAG_RESULTS message, which size depends on the amount of values to be returned and the size of this data type. If the end method (see page 96) returns true, the slave has ended its simulation and notifies this event by sending a TAG_FINALIZE message, otherwise the slave sends a TAG_RESULS MPI message.

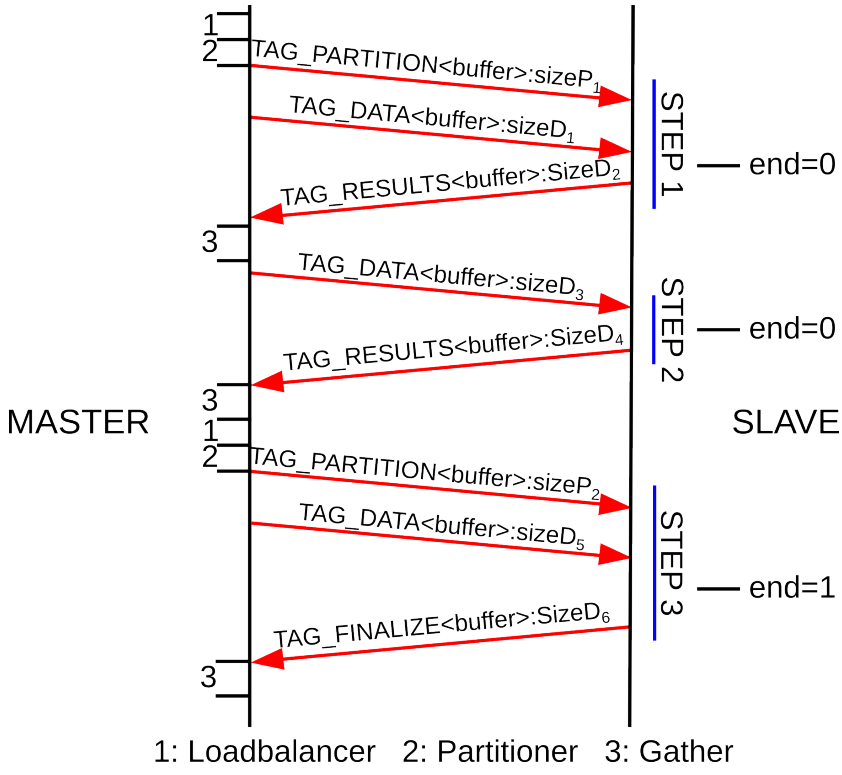


Figure 5.6: Execution phase timeline. This phase covers from the TAG_PARTITION message until the reception of the tag TAG_FINALIZE.

Termination phase: The termination or closing phase (see figure 5.7) includes the communication closing procedure. It starts with the tag `TAG_FINALIZE` sent by the master in broadcast to all the slaves and finish with the reception of the `TAG_END` sent in unicast to the master process by the slaves. Both messages occupy zero bytes as there is no need to send information with this message, only the notification of the end of the simulation. Once the slave sends the `TAG_END` message, it can free the memory used for its simulation and die as a MPI process it is, and once the master collects all the `TAG_END` tags, it can also free its memory and die.

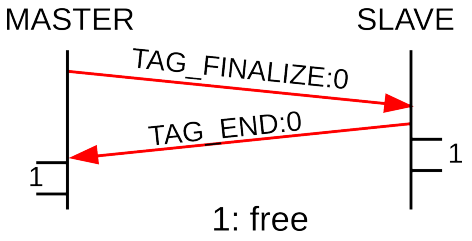


Figure 5.7: Closing phase. This phase covers the communication closing procedure from the tag `TAG_FINALIZE` until the reception of `TAG_END`.

In short, the following tag messages (see table 5.1) are sent and receive by both the master process and the slaves processes during the execution of an application using the proposed framework.

MPI TAG	$\rightarrow$	Type	Size	Transmission	Explanation
MPI_JOIN	$S \rightarrow M$	MPI_CHAR	255	Unicast	Notification of a new slave
MPI_CONFIGURE	$M \rightarrow S$	MPI_INT	C	Broadcast	Problem dimensions and directions
MPI_PARTITION	$M \rightarrow S$	MPI_INT	P	Unicast	Encapsulation of the partition
MPI_DATA	$M \rightarrow S$	MPI_CHAR	D_1	Unicast	Data encapsulation
MPI_RESULTS	$S \rightarrow M$	MPI_CHAR	D_2	Unicast	Values calculated by the slave
MPI_FINALIZE	$S \rightarrow M$	MPI_CHAR	D_2	Unicast	Execution of the last step
MPI_FINALIZE	$M \rightarrow S$	*	0	Unicast	ACK of the finalization of the slave
MPI_END	$M \rightarrow S$	*	0	Broadcast	Termination signal, ready to free

Table 5.1: Tags used by the master and the slaves processes. C is the size of the buffer where the offset of each configured direction and the dimensions of the problem are written, P is the size of the buffer where the control parameters of the partitions are written (see expression 3.4a), D_1 is the size of the data buffer sent to a slave process and D_2 is the size of the data buffer return from a slave process (see expression 3.4b). M stand for *Master* and S for *Slave*.

5.2 Implementation of a test application

Once the most important parts and modules of the framework have been reviewed, the details to implement and application that is integrated with the framework can be introduced.

As an example, in the following sections the implementation of a quite simple test application will be explained. This application, which name is `testApp`, will calculate, for each cell of the lattice, the average between the values of its first neighbours from the point of view of the von Neumman neighbourhood (see figure 6.2(a)). This average value is stored in a double variable. Although it is possible to use the default plugins provided by the platform, in this case the example `testApp` application will make use of the non-standard plugins and implements its own ones.

5.2.1 Implementation of the user datatype

As the platform is a completely generic framework, the user must specify and implement the datatype that best fits with its algorithm, that is, the user has to specify the datatype that will be stored in the vertex of the application graph. This datatype can be as complex as the user wants, since the process of this object will be made from the user code. At the moment, it is mandatory, due to compile reasons, to name the class as “Data”.

Defining the Data class

The datatype specified must always be included in the namespace `Reconfiguration` (see page 58) and inherit from the `ISerializable` pure virtual class (see code 3.19) since it is the only class that allows to send and receive an object through the network, by different processes in the cluster At this level, it is also necessary to specify the constructors and destructor of the data object, so that this object can be created and deleted by the system. Related to this point, it is recommended to define the empty constructor to initialize the object to its default values and the definition of the copy constructor that makes a new object of the class and sets the values from the original one, making a copy of it and avoiding its destruction.

Due to compilation issues, it is recommended the use of the preprocessing macros (refer als to macro) as `#ifdef` and `#endif` to protect the compilation process against a possible double definition of the class `Data`.

The final code of the *Data.h* can be seen in code 5.1.

```
1  #ifndef DATA_TESTAPP_H_
2  #define DATA_TESTAPP_H_
3
4  #include "ISerializable.h"
5  #include <iostream>
6  #include <sstream>
7
8  namespace Reconfiguration
9  {
10     class Data : public ISerializable
11     {
12         public:
13             double dValue; // Allocates the average of its neighbours
14
15             Data(); // Empty constructor
16             Data(double); // Normal constructor
17             Data(const Data&); // Copy constructor
18             virtual ~Data(); // Destructor
19
20             void print(); // To debug
21
22             // Deserialisation
23             std::stringstream& operator<<(std::stringstream&);
24
25             // Serialisation
26             std::stringstream& operator>>(std::stringstream&);
27     };
28 }
29
30 #endif /* DATA_TESTAPP_H_ */
```

Code 5.1: `testApp` datatype class definition.

Implementing the Data class

There are not changes regarding the normal operation of ANSI constructors and of the constructor of the user class.

The `print()` method will be called only if the framework is executed in `debug` mode. Thus, this method must be defined but may not be implemented.

As the `Data` class is something external and unknown for the framework and there is not a method to know which fields form the class nor to calculate a priori the size of an object, there is no way to binary dump the user data object into a network buffer nor the opposite action, restore an object from a network buffer. Thus, the user has to calculate the proper size of each field of the object and the size of the entire object in memory and, due to this situation, the serialisation and deserialisation of must be implemented by the user.

The serialisation and deserialisation methods are implemented by overloading the operators $>>$ and $<<$ respectively. To serialize an object means to dump the fields and values of the object into an array of bytes, whereas its deserialisation implies the restore of the object from the array of bytes. The input parameter of both methods is the buffer (byte `streamstring`) where the information has to be read or written; the object to serialize or deserialise is implicit in the call and references by the object.

The final code of the *Data.cpp* can be seen in code 5.2.

5.2.2 Implementation of the algorithm

The algorithm implemented by the user inherits from the `Algorithm` abstract class (see code 3.13), in order to let the framework and the user application to communicate with each other.

Defining the algorithm

The next step is define the algorithm class (see code code 5.3), which methods the user uses to solve or simulate his problem within the platform. This class can be as complex as the user wants and can have as many methods and variables as needed but it always must inherit from the public abstract class `Algorithm`.

Implementing the algorithm class

As the user algorithm inherits from the abstract `Algorithm` class, it must have implemented, at least, those methods called by callback from the framework. In general terms,

```
1  #include <iostream>
2  #include <limits>
3  #include "Data.h"
4
5  namespace Reconfiguration
6  {
7      Data::Data(){dValue = -std::numeric_limits<double>::max();}
8      Data::Data(double d){dValue = d;}
9      Data::Data (const Data& data){dValue = data.dValue;}
10     Data::~Data(){}
11
12     void Data::print(){std::cout << dValue;}
13
14     std::stringstream& Data::operator<<(std::stringstream& from)
15     {
16         from.read((char *)&dValue, sizeof(double));
17         return from;
18     }
19
20     std::stringstream& Data::operator>>(std::stringstream& to)
21     {
22         to.write ((char *)&dValue, sizeof(double));
23         return to;
24     }
25 }
```

Code 5.2: `testApp` datatype class implementation.

these functions extends the functionality of the framework with the code implemented by the user and modifies the behaviour of the platform to adapt it to the one specified by the user. Some of those functions are executed by the supermaster process whereas others, by the slaves.

The methods, that must be implemented at the algorithm class (see code 5.4), are the following ones:

`Algorithm` constructor: that initialises the variables needed to check the end condition and those ones defined by the user. This method must be called from the master and the slave side.

`Algorithm` destructor: frees the possible variables allocated by the user. It is also called by the master and the slave processes, since the memory was allocated by the constructor at the master and at the slave execution.

`Process` method: implementation of the logic for reading the initial values of the problem. This method receives between others, the input filename where the initial values are stored. The format of the input file must be in concordance with how this method reads the input values. It must read the corresponding values in order to create the `Data` objects by calling the `Data` constructor.

```
1  #ifndef TESTAPP_H_
2  #define TESTAPP_H_
3
4  #include "Data.h" // <--
5  #include "Errors.h"
6  #include "Graph.h"
7
8  // End condition
9  #define MAX_STEPS 15
10
11 namespace Reconfiguration
12 {
13     class testApp : public Algorithm
14     {
15         int iSteps;
16         public:
17             testApp();
18             virtual ~testApp();
19
20             int process (const char*, Data**, int);
21             void kernel(Graph **);
22             void save (Graph *, std::stringstream *);
23             //void postGather (Graph **);
24             int end(Graph *);
25     };
26 }
```

Code 5.3: `testApp` class implementation. The method `postGather` may or not be defined.

`Kernel` method: plays the proper role of the algorithm to be run on the cluster. It should always receive as an argument the graph on which to act and create a copy of it, to prevent or secure the copy received as reference. The object graph can be easily managed thanks to the overload of operators as "()" and "=": $()$ operator allows to manage the graph as it were a matrix structure, regardless of its dimensions (refer to page 39); $=$ operator makes a copy of the whole object by calling the copy constructor of the class `Graph` and on cascade, the copy constructors of the fields of the `Data` class, instead of doing a pointer assignment as this operation as usual does.

`Save` method: since the supermaster process does not contain any information about the `Data` class, it does not know which object fields must be stored and how to access to them; The only way to manage this situations is asking the user to implements its own procedure to identify which values must be saved and how. This situation is very similar to the one that justifies the user code for serialising and deserialising the `Data` object, but since serialisation and deserialisation works with the object as a whole, the save method transforms only the fields to be saved.

`End` method: This function checks if the slave process has finished its execution fulfilling the end condition. The only thing to consider is that this function should return "1" when it reaches the ending condition imposed by the user.

Sometimes it is useful to operate with the complete graph once the master process has gather the results of each slave into the new graph object. For these cases, the `postGather` method can be implemented. If the user does not define this method, the target of the callback will be, due to the inheritance feature of the *Object Oriented Programming*, an empty method of the `Algorithm` abstract class.

5.2.3 Implementations and use of the plugins

There are four categories of plugins, with each of them, the user can choose between implementing his own codes or using the ones provided by the framework. These categories are related to the four behaviours that the user can modify to adapt the global functionality of the framework to the one that best fits with the algorithm to execute.

The four categories are: plugins for the load balancing, plugins for the domain decomposition, plugins to save the results and plugins for grouping the nodes of the cluster.

Defining the header file

Depending on the category of the plugin, each one must inherit from a different class and thus, implement different methods to interact with. Once again, the plugin can be as complex

```
1  #include "Algorithm.h"
2  #include "Data.h"
3  #include "Errors.h"
4  #include "Exception.h"
5  #include "Framework.h"
6  #include "Graph.h"
7  #include "testApp.h" // <--
8  #include "Vector.h"
9
10 #include <cstring>
11 #include <fstream>
12 #include <iostream>
13 #include <stdio.h>
14
15 namespace Reconfiguration
16 {
17     // Executed by the supermaster and the slaves processes.
18     testApp::testApp)
19     {
20         iSteps=0;
21     }
22
23     testApp::~testApp()
24     {
25     }
26
27     // Executed by the supermaster process.
28     int testApp::process(const char *sName, Data **values, int n)
29     {
30         std::ifstream file;
31         double dValue;
32         int i=0;
33
34         if (!sName || !values || n<=0)
35             return INVALID_ARGUMENTS;
36
37         file.open(sName);
38         if (!file.is_open())
39             return NO_SUCH_FILE;
40
41         while (!file.eof() && i<n)
42         {
43             file >> dValue;
44             values[i] = new Data(dValue);
45             i++;
46         }
47
48         file.close();
49
50         if (i!=n)
51             return INVALID_SIZE;
52         return OK;
53     }
```

```
55      // Executed by the slaves processes.
56      void testApp::kernel(Graph **data)
57      {
58          Vector<int> vElements = (*data)->elements();
59          Vector<int>::iterator vi;
60
61          Graph* data2 = new Graph(**data);
62
63          for (vi = vElements.begin(); vi != vElements.end(); ++vi)
64              (*data2)(*vi)=((**data)(*vi, "N") + (**data)(*vi, "E") +
65                             (**data)(*vi, "S") + (**data)(*vi, "W")) / 4. ;
66
67          delete *data;
68          *data = new Graph(*data2);
69          delete data2;
70      }
71
72      // Executed by the master or the slaves processes.
73      void testApp::save(Graph *data, std::stringstream *ss)
74      {
75          int iValue;
76          int iElements = data->vVertex.size();
77
78          ss[0].write((char *)&iElements, sizeof(int));
79          for (int i=0; i<iElements; i++)
80          {
81              iValue = (int)(*data)(i).dWeight;
82              ss[0].write((char *)&iValue, sizeof(int));
83          }
84      }
85
86      // Executed by the slave processes.
87      int testApp::end(Graph *data)
88      {
89          iSteps++;
90          return (iSteps == MAX_STEPS);
91      }
92  }
```

Code 5.4: `testApp` class implementation.

as the user needs but it must fulfil different constraints in order to allow the communication between the plugin and the framework. The minimal conditions to fulfil are specified in the table 5.2.

Plugin category	**Inherit from**	**Implements**	**Communication with the framework through**:
Domain decomposition	ISorting	Constructor Destructor .so symbols sort method	$int * iSort$
Grouping	IGrouping	Constructor Destructor .so symbols group method	$Vector < Graph > vGraph$
Loadbalacing	IPlugin	Constructor Destructor .so symbols get method	$Vector < double > v$
Output	IFormat	Constructor Destructor .so symbols save method	–

Table 5.2: Specifications for the definition of the plugins.

The implementation of the functions to access to the symbols `New` and `Delete` must be done in an external C function. The integration of the plugin in the platform must be made using the cited variable, except for the output plugin. From the point of view of its management, the output plugins are a little bit special: unlike the other plugins, the user interaction with the framework ends in the plugin code so, in this case, the plugins does not need to send information to the framework to continue with the user request.

The code 5.5 shows how the `testPlugin` load balancing plugin must be defined. The same changes can be applied to other plugins from other categories taking into account the information of the table from above.

Implementing the plugins

For the implementation of a special plugin, regardless from its category, it must implement the corresponding methods, taking also into account the variable and its datatype that allows the communication with the framework, as shown in table 5.2. Any other methods or even object fields can be defined but they always remain internal to the plugin and unmanaged from the point of view of the platform.

As an example, the code that implements the `testPlugin` load balancing plugin can

```
1  #include "IPlugin.h"
2  #include "Vector.h"
3
4  namespace Reconfiguration
5  {
6      class testPlugin : public IPlugin
7      {
8          public:
9              testPlugin();
10             virtual ~testPlugin();
11             void get();
12     };
13
14     extern "C" IPlugin* New() {return new testPlugin;}
15     extern "C" void Delete(IPlugin* p) {delete p;}
16 }
17 #endif /* TESTPLUGIN_H_ */
```

Code 5.5: Definition of the `testPlugin` load balancing plugin and specifications for the `New` and `Delete` symbols

```
1  #include "Graph.h"
2  #include "IPlugin.h"
3  #include "testPlugin.h" // <--
4  #include "Timer.h"
5
6  namespace Reconfiguration
7  {
8      testPlugin::testPlugin():IPlugin(){}
9      testPlugin::~testPlugin(){m_v.clear();}
10
11     void testPlugin::get()
12     {
13         for (unsigned int i=0; i<m_graph->vVertex.size(); i++)
14         {
15             double dMemory = m_graph->vVertex[i].machine->dMemory;
16             m_v.push_back(dMemory * m_dWeight);
17         }
18     }
19 }
```

Code 5.6: `testPlugin` class implementation

be shown in code 5.6.

5.2.4 Upper layer

The main method is a standard one and it is the one responsible for providing a new `Algorithm` object by invoking its constructor method.

In the frame 5.7 the code of the main method is displayed. The main method will be executed by all the processes launched, regardless of its category: supermaster process, master processes or slave processes, with the highlight that all of them will instantiate the whole framework but they will only use those methods where their context make sense.

```
94  using namespace Reconfiguration;
95
96  int main (int argc, char *argv[])
97  {
98      try
99      {
100         Algorithm *algorithm = new testApp();
101         Framework framework(argc, argv, algorithm);
102         delete algorithm;
103     }
104     catch (baseException *e)
105     {
106         e->print();
107         return -1;
108     }
109
110     return 0;
111 }
```

Code 5.7: Main method implementation.

Some important points that must be highlighted:

- The `main` method must catch the possible exceptions thrown by the modules of the platform, but as the class `Exception` is an abstract class, they can only be catch by reference.
- In order to avoid memory leaks, the framework must be freed.
- If the framework is called from, for example, a script file, it must be needed to check the return value of the global execution. Due to this reason and fulfilling the standard error codes, the platform will return -1 in case of error or 0 otherwise.

5.3 Configuration files

There are four configuration files that the user must write before executing the application with the platform. This configuration files are related to the four features of the platform that can be modified by the user. The name of the four configuration files is passed as arguments to the executable and this way, the name of the file can be arbitrary.

5.3.1 Cluster configuration file

In this file (see code 5.8), the user defines the topology of the system graph by defining its vertices (nodes) and its edge (network connections between nodes). Both structures follow a fixed definition:

Node: Each machine of the cluster is specified by a structure called *node*. This node is identified by its hostname. It may also be possible to define the amount of memory (in Mb) that this node can use and its CPU frequency (in MHz). Those parameters will be used by the load balancing plugins if the corresponding plugins are loaded.

Link: The nodes of the system graph are connected to each other by edges, structures that specifies which two nodes are hooked up and the weight of this connection. This weight might be defined or not. It measures how difficult is to send a buffer through this link, in terms of time (in ms), bandwidth... This value could be related to the ping answer time but it is not yet discovered in real time.

```
1  node {
2    name: "agamemnon"
3    memory: 1024
4  }
5
6  node {
7    name: "a01"
8    cpu: 300
9    memory: 1024
10 }
11
12 link {
13   source: "agamemnon"
14   destination: "a01"
15   weight: 2
16 }
```

Code 5.8: Definition of a cluster with two nodes, *agammenon* and *a01* with 1Gb RAM and the second one runs at 300MHz, connected by a link of weight 2ms.

5.3.2 Load balancer configuration file

As it has been explained before, the user can specify its own load balancing plugins or use the ones provided by the platform but, even the first or the second option is chosen, the set of load balancing plugins to load at the beginning of the execution of the platform must be somehow specified. This "somehow" implies the use of the loadbalancer configuration file (see code 5.9).

```
1  plugin {
2    name: "Frequency"
3    weight: 0.15
4  }
5
6  plugin {
7    name: "Memory"
8    weight: 0.15
9  }
10
11 plugin {
12   name: "Previous"
13   weight: 0.7
14 }
```

Code 5.9: Load balancer configuration file. Three different plugins have been defined: the Frequency plugin, that contributes with 70% of the load balancing estimation; the Memory plugin, with 15% and the Previous plugin, that weight the previous execution time with 15%

The structure of the loadbalancer configuration file is divided into several objects called *plugin* having as many of these structures as plugins must be loaded. Each of these plugins structures must define:

- the name of the plugin structure, string that is also related to the name of the library defined by the user; if the user specifies that the plugin which name is "A" will be used, the framework will try to load the "libA.so" library.
- the weight of the plugin over the global set of plugins, given as fraction of unity.

5.3.3 Partitioner configuration file

The partitioner configuration file (see code 5.10) is divided into the definition of the heuristic function and the partitioner method:

Heuristic function: Determines when the partitioner algorithm has to be carried out. The execution of the heuristic function can be triggered by a constant number of steps,

following a lineal or exponential function, by thresholds, or even following a simulated annealing tendency.

Partitioner method: Specifies the algorithm that decompose the entire domain into several partitions. This method is identified by its name in the same way as in the case of the load balancing plugins. The name here specified will be concatenated with a fixed prefix and suffix, in order to obtain the real name of the dynamic library. A "variant" field can be also specified, value to specify whether the partition shape can change while the execution of the algorithm, that is, if the partition shapes is determined only one per execution of on every step that fulfils the heuristic function. The platform, as its own, provides three partitioning algorithms: the BFS method, the DFS method and methods based on space filling curves.

```
1  heuristic {
2    constant {
3      steps: 1
4    }
5  }
6
7  method {
8    type: DYNAMIC
9    name: "BFS"
10 }
```

Code 5.10: Domain decomposer configuration file. The application will execute the reconfiguration of the partition shapes after each step of the user algorithm. The algorithm used to calculate the partition is, in this example, the BFS algorithm.

5.3.4 Problem configuration file

The problem configuration file (see code 5.11) is used to define the properties of the user algorithm that will run in the cluster.

This is the most complex configuration file used by the platform, due to the amount of options and variants that can be defined to concrete the behaviour of the algorithm implemented by the user.

name: Name of the application to run. It may seem strange the fact of specifying the name of the application in the application configuration file but in this case, it is needed to give a name to the output files, as for example the log file, the execution time files and the result files.

cells: the application graph can be defined as a set of cells that have several connections

to other cells. These connections can be relative in 2D to a square lattice (the most common one) or even to triangular or hexagonal lattices, that is, those shapes that can tessellate the 2D space; for higher dimensions (3D, 4D...), more complex lattices can be defined. The cell structure defines then the set of neighbours that the vertices of the application graph have. For each type of lattice (in 2D are square lattices, triangular lattices and hexagonal lattices) the set of target coordinates must be defined, taking into account that, for example, in 2D, two coordinates must be defined in order to specify the connection between the actual cell and the target one. Each connection, defined by its coordinates, must be identified by an unique name.

topology: defines the vertices and links of the application graph, the amount of vertices and how are they arranged in a N-dimensional space. Each vertex of the application graph has a set of connections to link with other vertices. This set of connections is commonly called *communication pattern*. The communication pattern can be the same for all the vertices of the graph, that is, an homogeneous communication pattern but sometimes, due to symmetries of the application graph, it could be better to define heterogeneous communication patterns, different set of connections depending of the actual vertex. The application graph might be defined as a *matrix* if the communication pattern is homogeneous or as a *graph* if it is heterogeneous. In both cases, the connections between vertices are defined as directions, making use of the connections defined under the *cells* structure explained before. A mixed solution between matrices and graphs can also be adopted, making easy and faster the task of defining topologies were homogeneous pattern can be mainly found but there are still some parts of the graph were an heterogeneous pattern is applicable.

input: Name of the input file, where the initial values of the vertices of the graph are specified. The path of this file is relative to the directory where the binary is.

output: this structure specifies which output files are going to be obtained once the applications ends its execution. There is a substructure called *files* which refer to the new output files that are going to be saved at the end of each step of the algorithm, such as for example, the results of each step. In this substructure, the user must specify the plugins to be used to save the results, as for example, plain text TXT files or even PNG, JPG... image files.

```
1  name: "testApp"
2
3  input {inputFile: "testApp.problem"}
4
5  cells {
6      square {
7          coordinate {
8              name: "N"
9              distance: 0
10             distance: 1
11         }
12         coordinate {
13             name: "E"
14             distance: 1
15             distance: 0
16         }
17         coordinate {
18             name: "S"
19             distance: 0
20             distance: −1
21         }
22         coordinate {
23             name: "W"
24             distance: −1
25             distance: 0
26         }
27     }
28 }
29
30 topology {
31     dimension: 2
32     dimensions: 4
33     dimensions: 4
34     matrix {
35         boundary: true
36         boundary: false
37         all {
38             direction: "N"
39         }
40         all {
41             direction: "E"
42         }
43         all {
44             direction: "S"
45         }
46         all {
47             direction: "W"
48         }
49     }
50 }
51
52 output {
53     timeFile: "testApp.time"
54     logFile: "testApp.rms"
55     partitions: true
56     file {
57         name: ""
58         type: "PNG"
59     }
60 }
```

Code 5.11: Problem configuration file

5.4 Filesystem and compilation

If the test application has been implemented and configured, it can be now compiled and run. For compiling the application, this must be integrated with the framework not only from the computing point of view but also from the point of view of the filesystem. The integration of the applications within the platform lets it to find the binary file, the configuration files or even the plugins.

As it has been explained before (see page 14), the applications are organized as schemas and, in order to integrate an application with the platform, the corresponding schema must exist. The schema is not more than a profile that determines a set of folders organized in a fixed way. Integrating an application with the system means the automatic creation of the makefiles needed to compile the application and the creation of the filesystem, where for example, the source code must be allocated or the plugins stored. Deleting a schema implies the deletion of the profile, removing the application, the source code, the configuration files, the results achieved (if any) and the plugins from the filesystem.

The implemented user application is compiled against the framework by a generic makefile that invokes internal makefiles on cascade. These makefiles compile besides the application source code, the different plugins implemented by the user or even the standard ones, creating the corresponding dynamic libraries.

The filesystem is organized as follows as shown in box 5.12. At the root folder of the framework several folders are created: the documentation folder, with the man pages and *pdf* files; the *framework* folder, that stores the source code of the platform; the *include* folder, where all the header files are allocated; the *schema* folder, with the modules to integrate new application into the platform and to delete applications are; and apart from these folders, new folder named as the application is created. Inside this last folder, by default four more subfolders are created: the *bin* folder, where the binary and the configuration files reside; the *lib* folder, where the different `.so` files must be stored as a result of its source code; the *rst* folder, where the results of the execution are going to be saved and the *src* folder, where the user has to save the source code of the application to run. All of the folders here described have its own `makefile` to compile the corresponding files of the folder.

```
|-- testApp
| |-- bin
| | |-- cluster.conf
| | |-- testApp
| | |-- testApp.conf
| | |-- hostname
| | |-- loadbalancer.conf
| | `-- partitioner.conf
| |-- lib
| | |-- grouping
| | | |-- .cpp files
| | | |-- .h files
| | | |-- .so files
| | | `-- makefile
| | |-- loadbalancer
| | | |-- .cpp files
| | | |-- .h files
| | | |-- .so files
| | | |-- makefile
| | |-- makefile
| | |-- output
| | | |-- .cpp files
| | | |-- .h files
| | | |-- .so files
| | | |-- makefile
| | `-- partitioner
| | |-- .cpp files
| | |-- .h files
| | |-- .so files
| | `-- makefile
| |-- makefile
| |-- rst
| | `-- gifs.sh
| `-- src
| |-- Data.cpp
| |-- Data.h
| |-- testApp.cpp
| |-- testApp.h
| `-- makefile
|-- framework
| |-- .cpp files
| |-- .proto files
| |-- makefile
|-- include
| |-- .h files
| |-- Data.h -> ../testApp/src/Data.h
|-- makefile
|-- schema
| |-- add
| |-- del
| `-- replace
`-- Utils
    |-- batch
    `-- CreaMatriz.c
```

Code 5.12: Filesystem of the platform organisation

II

Tests and Results

6 Tests

Several tests have been developed to check the correctness and performance of the proposed framework. These test are presented here, ordered by its complexity, starting with the most simple one and finishing with the two most complex simulations; from the simple application for shifting values of the graph to computational fluid dynamics simulations and Sudoku puzzle solvers.

Each simulation or test can be categorized as a massive-computation (high rate of computations) application, as a massive-networking (high interaction between processes) application or even as a combination of both of them. Massive-computation will accomplish a better performance of the parallelisation of the application because, as the master process executes also complex algorithms to decompose the problem into domains and to balance the computation between slaves, the global runtime load imbalance will be reduced, maximizing the global performance.

This chapter is organized as follows: each section contains, to get involved with the problem to be solved, a first quite descriptive introduction to the simulation implemented. After that, the methodology and parallelisation subsection will be found, where the mathematical or functional model and the algorithm steps are specified. The second subsection, the code implementation, provides the specific information of how the simulation works: the user-data definition, the pseudocode of the kernel method (the one run by the slaves) and the neighbouring that must be defined in order to run the code making use of the framework.

6.1 Left Test Application

The application Left was the first application implemented and it was developed with the end of testing the most features of the early first version of the framework. Thus, this first application has no special scientific interest or relevance and will not contribute with any result to the thesis, only the assurance that the proof of concepts provides.

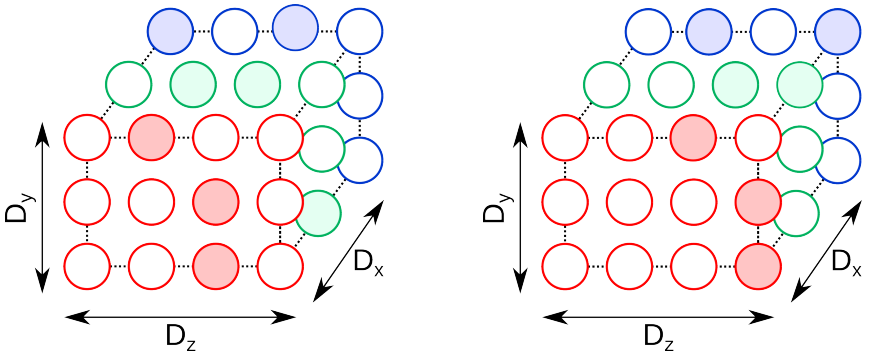
6.1.1 Methodology and parallelisation

The way this test application runs consists on a very simple step in which each cell of the lattice takes the value of its neighbour of its left, giving the impression that all the lattice values are moving to the right side, entering at the borders by the opposite extreme.

In order to describe the behaviour of the system and according to the expressions 3.2 and 3.3, the following expression can be applied:

$$g[i][j]\cdots[z] = g[i][j]\cdots[z-1], \forall i,j,\cdots,z \mid 0 \leq i < D_i, 0 \leq j < D_j, \cdots, 0 \leq z < D_z \quad (6.1)$$

the indices per dimension and where $D_i, D_j, \cdots, D_z$ are the sizes of the lattice per dimension (see figure 6.1).



(a) Before applying the algorithm (b) After applying the algorithm

Figure 6.1: Example of graph g before and after applying the `Left` algorithm. In this case, $D_x = 4$, $D_y = 3$ and $D_z = 3$

6.1.2 Implementation

The `Left` application runs over a $\Re^n$ Euclidean lattice and tests not only the communication and synchronisation of the involved processes, but also the domain decomposition algorithm and the idea of using plugins for the load balancing policy and the output files. The lattice defined has boundary continuous conditions (similar to the BvK conditions of physic models), that means, for a 2D lattice, the most right cells are connected to the ones located at the most left and the cells situated at the extreme positions of the up-bottom axis are connected with their opposite ones; so, in a 2D topology, the lattice is really a torus. This can also be applied to multidimensional lattices.

Data structure

The user data structure that allows to integrate the Left application into the framework consists on the one shown at code 6.1.

```
18      class Data : public ISerializable
19      {
20          public:
21              double dWeight;
22
23              Data();
24              Data(double);
25              Data(const Data&);
26
27              virtual ~Data();
28
29              void print();
30
31              std::stringstream& operator<<(std::stringstream&);
32              std::stringstream& operator>>(std::stringstream&);
33          };
34  }
```

Code 6.1: Data definition for the Left test application.

As seen before, the class must implement the *ISerializable* interface to being able to serialize and deserialise the user object and also must define the correct specific class constructor and destructor.

Neighbourhood

The application implemented to test the framework defines a very simple neighbourhood: each cell of the multidimensional lattice needs only the value of the cell of its left.

This very simple neighbourhood is useful to test whether the global space of the problem has been correctly decomposed and only the needed cells are updated step by step. A wrong behaviour of the framework would be easily recognized and debugged.

Kernel algorithm

The kernel algorithm executed by the slaves is described as pseudocode on 6.1.

This code is the simplest user code that the framework can execute.

```
name  : Left
input  : Graph representation of the domain received by the slave.
output: Graph representing the domain with new calculated values.
foreach node ∈ partitionElements do
  |  node.current ← node.atW
end
```

Algorithm 6.1: Pseudocode of the Left kernel algorithm applied to a $\Re^N$ space.

6.2 Game of Life Simulation Test Application

The game of life (commonly known as simply "life", "GoL (Game of Life)" or even "Conway's game of life") is a cellular automata (often termed as CA) devised by the British mathematician John H. Conway in 1970 [79]. "CA (Cellular Automata)" is a term first coined by the mathematician and pioneer of computer science, John von Neumann, to characterize a dynamical system that is discrete in space, time and state. Discreteness in space means that the dynamics occur on a lattice of cells; in time means that the cells update their states in a sequence of time steps; and in state refers to that each cell state can be described by a finite number of bits of information. At every step, each cell transitions to a new state that depends only on its current state and those of its neighbours according to a deterministic rule. The same rule is applied to each cell, simultaneously.

Many different types of cyclic patterns occur in the game of life, such as for example, "still lives", "oscillators" and "spaceships". These patterns have been used to test the correctness of the entire framework. The behaviour of these patterns is a well known matter, so the ones shown with the framework must be similar to one obtained with standard simulators.

6.2.1 Methodology and parallelisation

The standard GoL algorithm employs a two-dimensional Cartesian lattice and each cell of this one contains only can be only in one of two possible states. Following the metaphor of life, each cell is "alive" if its is equal to one or "dead" if its is equal to zero.

The rule to update the cell of the lattice rule is very simple and depends only on the current state of the cell and those of its eight neighbours (the four compass points and the diagonal directions). These rules are the following ones:

- If a cell is dead and has exactly three living neighbours, it becomes alive. As a metaphor, this event is called "birth".

- If a cell is alive, it will remain alive if it has either two or three living neighbours.
- If a cell has fewer than two living neighbours, it dies of “loneliness”.
- If a cell has four or more living neighbours, it dies of “overcrowding”.

These are the standard conditions and parameters but, as in many other cases, the problem can be easily parametrized. The standard GoL, in which a cell is “born” if it has exactly 3 neighbours; stays alive if it has 2 or 3 living neighbours; and dies otherwise, is symbolised as “B3/S23”. The first number is the requirement for a dead cell to be born. The second is the requirement for a live cell to survive to the next generation. Hence “B6/S16” means that a cell will born if there are 6 neighbours alive and it will live on if there are either 1 or 6 neighbours. On a two-dimensional grid, CA that can be described in this way, are known as Life-like CA. Some other variations modify the geometry (1D, 2D, 3D... and squared, triangular or even hexagonal lattice) of the universe as well as the basic rules. The basic rules may also be generalized such that instead of two states (live and dead) there could be three or more. State transitions are then determined either by a weighting system or by a table specifying separate transition rules for each state.

The initial pattern constitutes the seed of the system. The first generation is created by applying the above rules simultaneously to every cell in the seed. Each generation is the result of a pure function of the preceding ones. The rules continue to be applied repeatedly to create further generations until a number of generations is reached or an ending condition is satisfied.

According to the above definition, a cellular automata is completely deterministic. The way this simulation has been parallelised follows a simple “divide and conquer” philosophy. From the point of view of the slaves, they receive an independent domain, where the rules of the GoL must be applied. The way the domain decomposition algorithm treats the graph assures a decomposition into independent parts, avoiding the possible interactions between neighbour domains. The results of each slave are sent back to the master process, which gather the results in a new entire problem and starts a procedure to execute the next step of the algorithm.

Cellular Automata

In order to give a definition of a cellular automata, a very simple examples is presented [80]. This example was initially proposed by Edward Fredkin in the 1970s. The example uses a two dimensional square lattice and each site of it is a cell labelled by its position $\overrightarrow{r} = (i, j)$ where i and j are the row and column indices. A function $\psi_t(\overrightarrow{r})$ is associated to the lattice to describe the state of each cell at an iteration t. This quantity can be either 0 or 1.

The CA rule specifies how the states ψ_{t+1} are to be computed from the states at iteration t. The initial condition time is $t = 0$ with a given configuration of the values $\psi_0(\overrightarrow{r})$ on the lattice. The state at time $t = 1$ will be obtained as follows:

1.– Each site $\overrightarrow{r}$ computes the sum of the values $\psi_0(\overrightarrow{r})$ on the four nearest neighbour sites $\overrightarrow{r}$ at north, west, south and east. The system is supposed to be periodic in both i and j directions (torus topology) so that this calculation is well defined for all sites.

2.– If this sum is even, the new state $\psi_1(\overrightarrow{r})$ is 0 and else, it is 1.

The same rule (steps 1 and 2) is repeated over to find the states at time $t = \{2, 3, 4...\}$ calculated according to the previous states.

From a mathematical point of view, this CA parity rule can be expressed by the following relation:

$$\boxed{\psi_{t+1}(i,j) = \psi_t(i+1,j) \otimes \psi_t(i-1,j) \otimes \psi_t(i,j+1) \otimes \psi_t(i,j-1)} \tag{6.2}$$

where the symbol $\otimes$ stands for the exclusive OR logical operation. This is also equal to the sum modulo 2: $1 \otimes 1 = 0$, $0 \otimes 0 = 0$, $1 \otimes 0 = 1$ and $0 \otimes 1 = 1$.

This property of generating complex patterns starting from a simple rule is generic of many CA rules. Here, complexity results from some spatial organization which builds up as the rule is iterated. The various contributions of successive iterations combine together in a specific way. The spatial patterns that are observed reflect how the terms are combined algebraically. This example shows that despite the simplicity of the local rule, the global behaviour of a CA model can be quite complex. In the present case, the mechanisms yielding these complex patterns can be unravelled by working out how successive iterations combine several copies of the initial configuration, all shifted by a different amount.

This example gives a definition of a CA. Formally a cellular automata is made of:

- A regular lattice of cells covering a portion of a N-dimensional space.
- A set:

$$\boxed{\Phi(\overrightarrow{r},t) = \Phi_1(\overrightarrow{r},t), \Phi_2(\overrightarrow{r},t), \cdots, \Phi_m(\overrightarrow{r},t)} \tag{6.3}$$

  of boolean variables attached to each site $\overrightarrow{r}$ of the lattice and giving the local state of each cell at the time $t = \{0, 1, 2, ...\}$
- A rule $R = \{R_1, R_2, \cdots, R_m\}$ which specifies the time evolution of the states $\Phi(\overrightarrow{r},t)$ in the following way:

$$\boxed{\Phi_j(\overrightarrow{r},t+\tau) = R_j(\Phi(\overrightarrow{r},t), \Phi(\overrightarrow{r}+\overrightarrow{\delta_1},t), \Phi(\overrightarrow{r}+\overrightarrow{\delta_2},t), \cdots, \Phi(\overrightarrow{r}+\overrightarrow{\delta_j},t))} \tag{6.4}$$

where $\overrightarrow{r} + \overrightarrow{\delta_k}$ designate the cells belonging to a given neighbourhood of a cell $\overrightarrow{r}$.

The example 6.2.1 is a particular case in which the state of each cell consists of a single bit of information $\Phi(r,t) = \psi_t(\overrightarrow{r})$ and the rule is the addition modulo 2 ($\otimes$).

In that example, the rule R is identical for all cells of the lattice and is applied simultaneously to each of them, leading to a synchronous dynamics. It is important to notice that the rule is homogeneous, it does not depend explicitly on the cell position $\overrightarrow{r}$.

6.2.2 Implementation

Doing a comparison between the four rules that define the game of life and the example given in 6.2.1, a very simile can be observed. In this case, the $\Phi(r,t)$ expression follows the regular expression:

$$((state = 0) \wedge (alive = 3)) \wedge (((state = 1) \wedge (alive = 3)) \vee ((state = 1) \wedge (alive = 2))) \quad (6.5)$$

that is equal to:

$$(alive = 3) \wedge (state \otimes (alive = 2)) \quad (6.6)$$

where $(alive = x)$ means exactly x neighbours alive and $status$ is the state of the current cell. $ALIVE$ and $DEAD$ take the values 1 and 0 respectively. From a mathematical point of view, this expression gives the condition for the $ALIVE$ value and it has been obtained by operating over the rules cases shown in table 6.1.

	Number of alive neighbours								
State	0	1	2	3	4	5	6	7	8
0	DEAD	DEAD	DEAD	**ALIVE**	DEAD	DEAD	DEAD	DEAD	DEAD
1	DEAD	DEAD	**ALIVE**	**ALIVE**	DEAD	DEAD	DEAD	DEAD	DEAD

Table 6.1: Rules for updating a GoL cell

For a generic GoL, modelled by a BX/SYZ, the transformation rules can be expressed by the formula:

$$((state = 0) \wedge (alive = X)) \wedge (((state = 1) \wedge (alive = Y)) \vee ((state = 1) \wedge (alive = Z))) \quad (6.7)$$

Data structure

The user data structure that allows to integrate a sequential GoL simulation into the framework consists on the one shown at code 6.2.

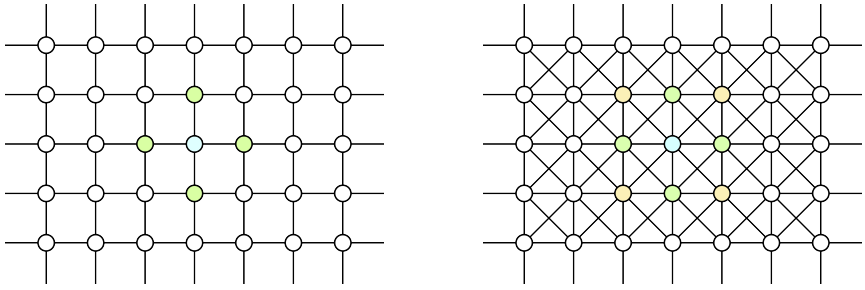
```
18      class Data : public ISerializable
19      {
20          public:
21              double dWeight;
22
23              Data();
24              Data(double);
25              virtual ~Data();
26              void print();
27
28              std::stringstream& operator<<(std::stringstream&);
29              std::stringstream& operator>>(std::stringstream&);
30      };
```

Code 6.2: Data structure definition for the NEP algorithm.

Neighbourhood

Boundary cells are a typical example of spatial inhomogeneities. The neighbourhood is usually made of the adjacent cells of the central cell. It is often restricted to the nearest or next to nearest neighbours, otherwise the complexity of the rule is too large.

For a two-dimensional CA, two neighbourhoods are often considered: the von Neumann and the Moore neighbourhood. The von Neumann neighbourhood (figure 6.2(a)) consists of a central cell (the one to be updated) and its four geographical neighbours North, West, South and East. The Moore neighbourhood (figure 6.2(b)) contains, in addition, the second nearest neighbours, that is, North-East, North-West, South-East and South-East, doing a total of nine cells (see figure 6.2).



(a) von Neumann neighbourhood (b) Moore neighbourhood

Figure 6.2: 2D-GoL Neighbourhood

For a N-dimensional CA, more and even more complicated lattices can be defined, most of them based on 2D neighbourhoods.

Kernel algorithm

The kernel code for simulate the GoL automata is described as pseudocode on code 6.2.

```
name : GameOfLive
input : Graph representation of the domain received by the slave.
output: Graph representing the domain with new calculated values.
foreach node ∈ partitionElements do
    living ← node.atE + node.atW + node.atN + node.atS + node.atNE + node.atNW +
    node.atSE + node.atSW
    if node.current = ALIVE then
        if living =2 ∨ living =3 then
            /* Still alive                                          */
            node.current =ALIVE
        else
            /* Loneliness or overcrowding                           */
            node.current =DEAD
        end
    else
        if living =3 then
            /* Birth                                                */
            node.current =ALIVE
        else
            /* Still dead                                           */
            node.current =DEAD
        end
    end
end
```

Algorithm 6.2: Pseudocode of the GoL kernel algorithm applied to a Moore neighbourhood.

It basically calculates the number of neighbours that are alive and applies the rules for determining the new value of the actual cell (table 6.1).

6.3 Potts Simulation Test Application

The Potts model [81] was described by Renfrey B. Potts in 1952 and it is a generalization of the Ising [82] model. The Ising model is a CA (see 6.2.1) that presents a magnetized material as a collection of spins arranged in a lattice. Each spin could only take two values (states), up or down (1 or -1 respectively). Potts generalized later the Ising model allowing Q states for each particle in the system.

In recent years, many different single-cell-based models have been developed and applied successfully to various biological and medical problems. These models use different computational approaches [83]. One of these models, developed by Graner and Glazier [84], is called the CPM (Cellular Potts Model), an extension of large Q Potts model, used to study differential adhesion and cell sorting in confluent sheets of cells. Some recent applications of the CPM are widely used (see [85, 86]).

The main idea focuses on the simulation of the Potts Ferromagnetic model, by the allocation of resources and tasks and the configuration of the topology of the network, by transferring the workload onto other computers of the cluster, to find dynamic matching between the tasks and the global resources of the network of computers, that optimizes this simulation completion time. This will be done assuming a non-static and decentralized approach.

This model has the ability to model thousand of individual cells in a certain space. Each cell has its own set of properties, such us size, type or adhesion strength to the neighbour cells. Cells are represented as extended objects on a square lattice and represented with an unique number (called spin in the large Q-Potts model) and a type. Adhesion between cell types is incorporated by defining surface energies between neighbour cell membranes.

The Ising and Potts models have a great computational efficiency disadvantage, due to their implementation using a Montecarlo method simulation, with a standard Metropolis algorithm. This algorithm needs hard computations to obtain the N-possible configurations for approaching to the system behaviour.

6.3.1 Methodology and parallelisation

The central component of the CPM is the definition of the Hamiltonian of the problem, determined by the configuration of the cell lattice. The lattice is usually taken to be a two-dimensional rectangular Euclidean lattice but it is often generalized to other dimensions or lattices. The Potts model consists of spins placed on that lattice.

The Hamiltonian function is defined as follows:

$$H = J \sum_{(i',j') \in Neigh(i,j)} 1 - \delta(\sigma_{(i,j)}, \sigma_{(i',j')}) \tag{6.8}$$

where J is a positive constant; δ is the Kronecker delta; $1 \leq \sigma_{(i,j)} \leq N_d$ denotes the orientation of the spin at (i,j); N_d is the number of possible domains in the system at the beginning of the simulation and $(i,j),(i',j')$ denotes the first neighbourhood area.

The probability function is defined as:

$$P = \begin{cases} e^{-\Delta H / k_B T} & \text{with } \Delta H > 0 \\ 1 & \text{with } \Delta H \leqslant 0 \end{cases} \tag{6.9}$$

where K_B is the Boltzmann constant and T is a certain temperature value. The new state of the spin is accepted with probability P (see expression 6.8).

6.3.2 Implementation

The evolution of the model proceeds using the Metropolis–Montecarlo method [87] simulation as follows:

1.– Select a random lattice spin.

2.– Choose randomly a neighbour spin of the current one.

3.– If the neighbour spin is equal to the spin of the current area, go to step 1.

4.– Change the current spin with the spin of its neighbouring area, according to the probability *P* as in expression 6.9.

5.– Return to step 1.

The lattice can be represented as a torus and is continually updated: for each lattice point, a different spin state is proposed and the new overall energy is calculated. The energy depends on the interactions between neighbours and the overall temperature of the lattice. If the new energy is smaller than the old one, the new state is accepted and the spin updated. If not, there is still a certain chance to be accepted, leading to random spin flips which represent the overall temperature.

The critical part of the algorithm, from the point of view of the parallelisation, resides in the step 2.

The parallelisation of CPM is an active area of research [83, 88, 89]. The critical part of the algorithm, from the point of view of parallelisation, resides in steps 2 and 4. *A priori*, the

domain and data decomposition does not always assure that the values needed to compute the calculus and new values are known by the processor that calculates that certain value (see figure 6.3).

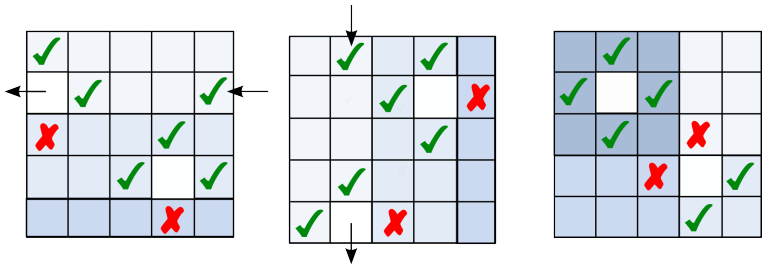


Figure 6.3: Depending on the domain decomposition policy, it can happens that the neighbours of a certain position of the grid are not placed in the same processor. Situation in which one processor must communicate with the neighbour processor to being able to update the current grid position.

The Potts model using the conventional Montecarlo method method has two drawbacks: the critical slowing down and the generation of random numbers. These drawbacks are the two computer time-consuming factors. Many parallel algorithms have recently been developed to reduce this "drawback" [88, 90]. The problem of these algorithms is that they are much more difficult to parallelise efficiently.

The goal is to parallelise the Potts model for being executed in a parallel environment, created by an heterogeneous cluster. The aim is to increase the speed-up and the efficiency to reduce these drawbacks across of a correct domain and data decomposition, assuring us that, on the one hand the needed values for the "spin-flip" attempt calculus are accessible by the processor that is calculating the possible spin-flip attempt and not have problem with a wait queue and, on the other hand, after of a MCS (Montecarlo Step) the domain decomposition is always correct.

The solution proposed describes an alternative parallel Metropolis–Montecarlo solution [91] more efficient than the traditional methods, using the framework explained before. The main difference between this algorithm and the non-parallel one is that this one divides the computation in several domains assigned each one to a MPI process. These processes could be of two different kinds: the slaves, that solve the problem and the master that only manages the synchronization of the different slaves processes. This election has been adopted in order to clearly separate the communication between MPI processes, the synchronisation with all the nodes of the cluster and the algorithm of the problem to solve. The communications between the master and the slave processes is made by the use of signals and tags to identify them.

Data structure

The user data structure that allows to integrate a sequential Potts simulation into the framework consists on the one shown at code 6.3.

```
18      class Data : public ISerializable
19      {
20          public:
21              double dWeight;
22
23              Data();
24              Data(double);
25              virtual ~Data();
26              void print();
27
28              std::stringstream& operator<<(std::stringstream&);
29              std::stringstream& operator>>(std::stringstream&);
30      };
```

Code 6.3: Data structure definition for the NEP algorithm.

Neighbourhood

For each cell located at a random position of a 2D Euclidean lattice, the set of adjacent cells needed to calculate the next value of that cell is expressed by the von Neumann neighbourhood (see figure 6.2(a)).

In case of simulating the Potts model in a more complex space, it must be fixed taking care of the Hamiltonian function 6.8 and adapting it to the new partition function.

The simulation performed considers a toroidal Potts lattice; it means that once selected a position from the lattice, its neighbourhood could be out of the simulation surface limits, but in the toroidal case, one atom situated on the limit of the simulation area has its neighbours at the opposite limit. This is only applicable to the master lattice being not true for the slave domains.

Kernel algorithm

The kernel code for solving the Potts problem is described as pseudocode on code 6.3.

```
name : Potts
input : Graph representation of the domain received by the slave.
output: Graph representing the domain with new calculated values.
foreach node ∈ partitionElements do
    newCell ← selectRandomCell ()
    (newSpin, energyDiff) ← calculateDeltaU (newCell)
    /* If energy decreases                                          */
    if energyDiff ≤ 0 then
        updateNodeSpin (newSpin)
    else
        /* Otherwise                                                */
        r ← randomNumber
        flipProbability ← e^(−energyDiff/K_B T)
        /* Still a chance to flip                                   */
        if r < flipProbability then
            updateNodeSpin (newSpin)
        end
    end
end
```

Algorithm 6.3: Pseudocode of the Potts kernel algorithm. See also 6.4 pseudocode.

6.4 NEP Simulator Test Application

NEP (Networks of Evolutionary Processors) [92] are a new computing mechanism directly inspired in the behaviour of cell populations. A NEP can be defined as a graph whose nodes are processors which perform very simple operations over strings sending them again to other nodes. The operations included in the basic model are well defined being these ones: erasing and adding a symbol, replacing a symbol with other, and splicing two string. Every node has filters that block some strings from being sent and/or received. As computing devices, NEPs alternate evolutionary and communicating steps, until a predefined stopping condition is fulfilled. All the processors change their contents at the same time in each evolutionary step. In the communicating steps, the strings, that pass the corresponding filters are written on the graph and the strings, that pass their input filters to build their contents for next steps, are taken from the graph.

In order to test the performance a cluster of computers when running parallel NEPs, a family of graphs to solve several instances of the HPP (Hamiltonian Path problem) have been designed. HPP can be solved by means of NEPs with a lineal (temporal) performance using a multi-threaded Java simulator [93] for NEPs. Although the goal is not to reach this bound but this proof will give us useful hints to improve the performance of the simulation of NEPs on non-parallel hardware platforms.

```
name  : calculateDeltaU
input  : Graph representation of the domain received by the slave.
input  : spinRandomCell selected by the slave.
output: newSpin value.
output: spinDiff between cell spins

temporalSpin ← currentSpin ()
leftSpin ← spinAtW ()
rightSpin ← spinAtE ()
topSpin ← spinAtN ()
bottomSpin ← spinAtS ()

if temporalSpin ≠ leftSpin then
  |  oldSpinDiff ++
end
if temporalSpin ≠ rightSpin then
  |  oldSpinDiff ++
end
if temporalSpin ≠ topSpin then
  |  oldSpinDiff ++
end
if temporalSpin ≠ bottomSpin then
  |  oldSpinDiff ++
end

newSpin ← newRandom ()
if newSpin ≠ leftSpin then
  |  newSpinDiff ++
end
if newSpin ≠ rightSpin then
  |  newSpinDiff ++
end
if newSpin ≠ topSpin then
  |  newSpinDiff ++
end
if newSpin ≠ bottomSpin then
  |  newSpinDiff ++
end

spinDiff ← newSpinDiff- oldSpinDiff
```

Algorithm 6.4: Pseudocode of the $calculateDeltaU$ function used by the Potts algorithm.

6.4.1 Methodology and parallelisation

This well-known NP–complete problem searches a Hamiltonian path in an undirected graph that is, the path that visits each vertex of the graph exactly once.

Hamiltonian path problem solution by NEPs

It is possible to solve this problem by means of the following NEP: a NEP graph is very similar to the one studied with an extra node to ease the definition of the stopping condition.

- Let n be the number of nodes of the graph under consideration (see figure 6.4).
- Let $\{v_i, 0 \leq i \leq n\}$ the set of processors of the NEP.
- The set $\{i, 0, 1, \cdots, n\}$ is used as the alphabet. Symbol i is the initial content of the initial node (v_0). Each node (except the final one) adds its number to the string received from the network.
- Input and output filters are defined to allow the communication of all the strings that have not yet visited the node. The input filter of the final node excludes any string which is not a solution.

It is easy to imaging a regular expression for the set of solutions (those words that fulfils the proper length, the proper initial and final node and where each node appears only once).

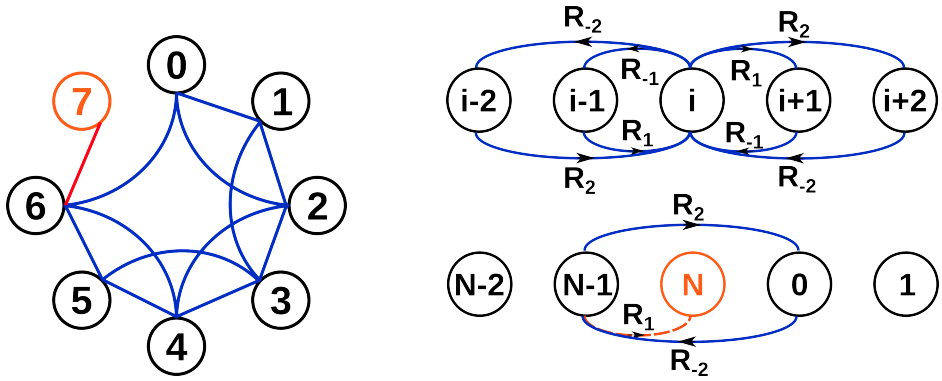
6.4.2 Implementation

The goal is to check the cluster performance solving the HPP for graphs of increasing difficulty. A family of graphs with n nodes have been used. 0 is the label of the initial node and each node is connected with the four closest nodes; that is, the node i is connected with the set of nodes $\{i-2, i-1, i+1, i+2\}$ if they exist. It is also needed to add an output node in this case, with the highest label given $(n+1)$. The output node is only connected with the final node of the graph under consideration n. Other connections of the output node are removed.

Depending on the connections made between the node i and its neighbours, they connections are called as follows:

- R_{-2}: connection between nodes labelled as i and $i-2$, if exists.
- R_{-1}: connection between nodes labelled as i and $i-1$, if exists.
- R_1: connection between nodes labelled as i and $i+1$, if exists.

- R_2: connection between nodes labelled as i and $i+2$, if exists.



(a) Example of a 6-NEP (NEP with n=6 elements) with the extra element to collect the output strings.

(b) Communication pattern between elements of a generic n-NEP.

Figure 6.4: NEP example and communication pattern of a NEP.

Data structure

The user data structure that allows to integrate the sequential NEP into the framework consists on the one shown at code 6.4. The basic datatype of this object is the vector of strings needed to store the string that each node has. As usual, the class must implement the *ISerializable* interface and define the correct specific class constructor and destructor.

```
21  class Data : public ISerializable
22  {
23      public:
24          std::string id;
25          Vector<std::string> contents;
26
27          Data();
28          Data(std::string id, Vector<std::string>);
29          virtual ~Data();
30          void print();
31
32          std::stringstream& operator<<(std::stringstream&);
33          std::stringstream& operator>>(std::stringstream&);
34  };
```

Code 6.4: Data structure definition for the NEP algorithm.

Neighbourhood

Due to the specific properties of the NEPs, these can be somehow modelled as a planar graph (1D graph) and consequently the domain decomposition for this problem is the easiest one we can find. As the extreme cells are connected between them, the NEP model can be considered as a cylindrical space.

Flattening the graph of the figure 6.4(a), the planar graph of figure 6.4(b) is obtained. Each element i of the NEP, except the output one that collects the results and the ones which has "virtual" connections with this one, has as adjacent elements its first order neighbours (nodes $\{i-1, i+1\}$) and its second order neighbours (nodes $\{i-2, i+2\}$).

Kernel algorithm

The algorithm, that the NEP problem follows, is described as pseudocode on code 6.5. Basically, each slave takes the content strings of its neighbours and concatenates each string with its own identifier. So, a string like $0 \rightarrowtail 2 \rightarrowtail 4$ received by the element 5 of the NEP 6.4(a) means that, from the initial node to the current one, exists a path that pass through the nodes 2 and 4. This string will be updated to $0 \rightarrowtail 2 \rightarrowtail 4 \rightarrowtail 5$ in this current node and will be sent to its neighbours.

This algorithm has two particular implementation issues. The first one is related to the behaviour of the initial node 0, due to its "heterogeneous" communication pattern (shown in figure 6.4(b)). This node can only receive information from its R_{-1} rule and can not send strings because it has no output rules. As it receives the results of the algorithm, it is the only node able to determine whether the strings are valid, checking for that the Hamiltonian path properties. On the other hand, the second implementation issue cited is related to the behaviour of the nodes that "break" the global communication pattern, in this case, the nodes 0, 1, $N-1$ and $N-2$. These nodes are not allowed to access to node N.

```
name : NEP
input : Graph representation of the domain received by the slave.
input : Number of nodes of the original NEP
output: Graph representing the domain with new values of the problem.

N ← NEPNodes
foreach node ∈ partitionElements do
    /* Special treatment for the last node of the NEP                    */
    if node = N then
        /* Apply input rule (access to node N − 1)                       */
        items ←+ apply(R_−1)
        foreach item ∈ items do
            /* Check hamiltonian path properties.                        */
            if itemNotValid (item) then
                rejectItem (item)
            end
            saveOuput
        end
    else
        /* Every other node of the NEP                                   */
        cleanItemsNode (node)
        /* Apply input rules (exclude access to node N)                  */
        if node ≠ 1 then
            items ←+ apply(R_−2)
        end
        if node ≠ 0 then
            items ←+ apply(R_−1)
        end
        if node ≠ N − 1 then
            items ←+ apply(R_1)
        end
        if node ≠ N − 2 then
            items ←+ apply(R_2)
        end
        foreach item ∈ items do
            addItem (concatenate (item, ´_nodeId´))
        end
    end
end
```

Algorithm 6.5: Pseudocode of the NEP kernel algorithm

6.5 Sudoku Solver Test Application

Sudoku, which is a Japanese term that means “singular number”, has gathered in this last decade much international popularity. The original Sudoku puzzle consists of a 9×9 grid with 81 Sudoku cells which are partitioned into 9 blocks of 3×3 numbers. Each puzzle has some cells that have already been filled in and some other empty ones. The objective of the puzzle is to fill in empty cells with the numbers 1 to 9 so that the following three rules are fulfilled:

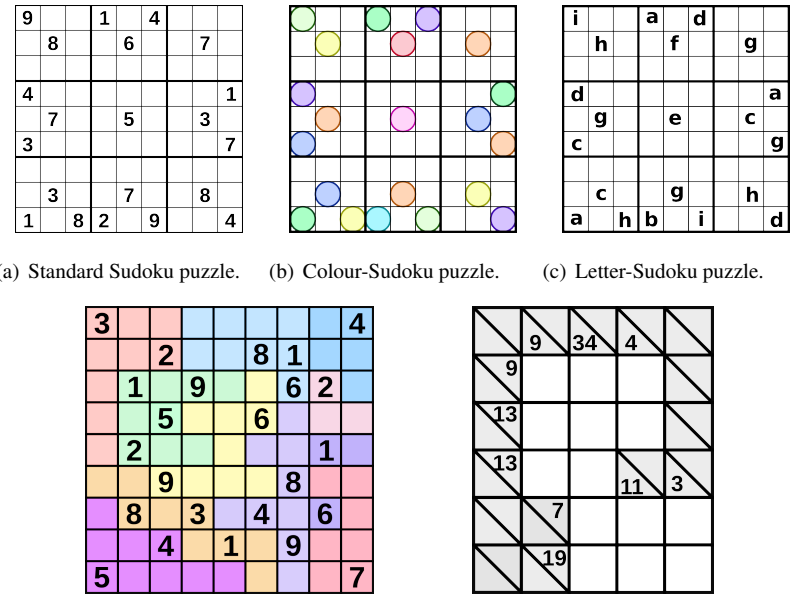
- Each horizontal row should contain the numbers $1 - 9$, without repeating any one.
- Each vertical column should contain the numbers $1 - 9$, without repeating any one.
- Each 3×3 block should contain the numbers $1 - 9$, without repeating any one.

Standard Sudokus (see figure 6.5(a)) are the ones explained above but, there is a wide variety of problems on literature based on these properties, adding or changing its basic rules, leading to more sophisticated and complex problems. A classification of puzzles attending to its properties and variation of its rules can be made:

- From the point of view of the information stored on each cell, several puzzles can be found, such as for example colour-Sudokus, filled in with colours instead of numbers (see figure 6.5(b)); letter-Sudokus, filled in with alphabetical values (see figure 6.5(c)); or even symbol-Sudokus, filled with generic symbols.
- From the point of view of the shape of the blocks, some other puzzles as Jigsaws, with non-homogeneous block shapes (see figure 6.5(d)), ken-ken or Kakuros, where the initial values are the result that must be obtained adding the set of values that fill the neighbours of the cell (see figure 6.5(e)), are also very common to find.
- The geometry of the sudoku can vary, such as it happens for example in Samurai-Sudokus.
- The dimensions of the Sudoku puzzle can be also parametrized.

In all cases, the set of symbols must fit the properties of the problem.

The application used for testing the framework consists on a parametrized Sudoku solver, able to run a sequential solver in a parallel and distributed way.



(a) Standard Sudoku puzzle. (b) Colour-Sudoku puzzle. (c) Letter-Sudoku puzzle.

(d) Jigsaw-Sudoku puzzle. (e) Kakuro-Sudoku puzzle.

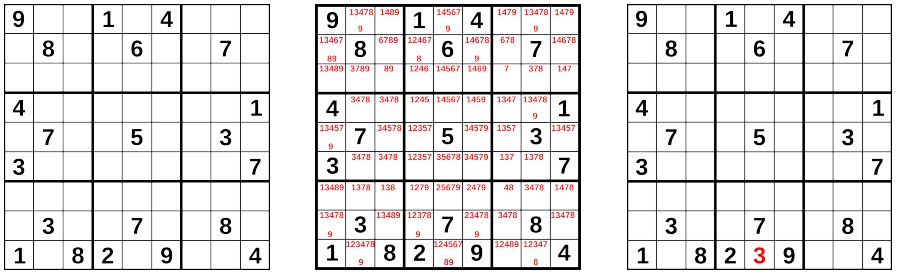
Figure 6.5: Puzzle variants based on the standard Sudoku rules.

6.5.1 Methodology and parallelisation

Each Sudoku cell stores information (cell values) in order to calculate the set of values that each of its neighbours of its domain can take. This information consists on a *forbidden list* with the values assigned on each cell of the domain.

An empty cell must be represented with an invalid value for the puzzle, in this case with the value 0. At the beginning of the algorithm, all the forbidden lists are initialized with the initial values of the cells of its neighbours (from the point of view of the domain decomposition) and once a cell is updated, the value of this cell is added to the forbidden list of its neighbours and its own forbidden list is cleared. In this way, the procedure to solve the Sudoku puzzle consists on generating the equivalent anti-Sudoku (see figure 6.6(b)) from a Sudoku puzzle step by step, where an anti-Sudoku is the puzzles that instead of having a fixed values on the cells, it has a set of values that can not fulfil the rules of the puzzle at a certain snapshot of the solving algorithm.

To set a value in a certain cell, the algorithm iterates over the forbidden list of it and selects a value that is not part of the list. If more than a value can be selected in this step, the algorithm has not enough information to opt for any one and leaves the forbidden list as it is.



(a) Initial Sudoku ($Sudoku1$) puzzle.

(b) Initial anti-Sudoku ($anti-Sudoku1$) puzzle obtained from $Sudoku1$.

(c) Sudoku ($Sudoku2$) puzzle obtained from $anti-Sudoku1$.

Figure 6.6: Example of an anti-Sudoku puzzle

The puzzle to be solved is decomposed into as many domains as slaves are involved in the resolution. Beginning with a valid Sudoku, an anti-Sudoku will be obtained by the slaves, writing the forbidden values of each cell of its partition. Once all the slaves have computed the anti-Sudoku of its partition and the master has received this information from all the slaves, it tries to choose a value that is not in the forbidden list of each of the cells, getting in this way the next Sudoku to iterate over (see figure 6.6).

6.5.2 Implementation

Since the main goal of this application is not to solve Sudoku puzzles, but it is only a communication-intensive application to test the functionalities and performance of the framework as a whole, the algorithm used has many limitations. For example, it is not able to solve cycles (N different blocks, each one with N cells with possible $\{x_1, x_2, ..., x_N\}$ values) or detect complex constrains. In general, it can resolve situations as "the only number of that position" but it can not solve constrains as "the only position for that number".

The algorithm used discovers the values that can not fill in a certain Sudoku cell, taking into account the properties of the Sudoku puzzle. The set of these numbers are stored in a forbidden list so, to determine a symbol that fulfils the three basic Sudoku properties, it is enough to select a number that is not in that forbidden list.

Data structure

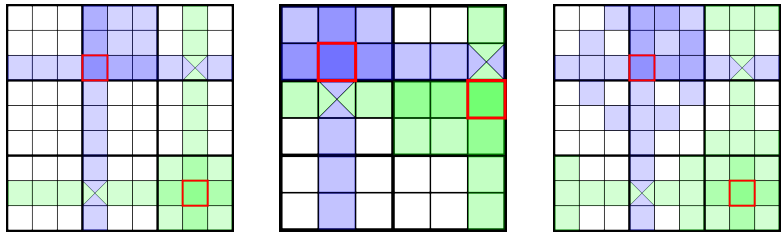
The user data structure that allows to integrate a sequential Sudoku solver into the framework consists on the one shown at code 6.5.

```
21      class Data : public ISerializable
22      {
23          public:
24              Vector<int> id;
25
26              Data();
27              Data(Vector<int> id);
28              virtual ~Data();
29              void print();
30
31              std::stringstream& operator<<(std::stringstream&);
32              std::stringstream& operator>>(std::stringstream&);
33      };
```

Code 6.5: Data structure definition for the Sudoku solver algorithm.

Neighbourhood

Taking into account the rules for fixing an item in a classical Sudoku (see page 130) each cell of the puzzle must know the values of all the cells of its row, column and block so, in this case, the problem must be initially decomposed into domains which satisfy that all the elements of a row, column or block must stay together in the same process (see figure 6.7).



(a) Classical Sudoku geometry (b) Non-classical Sudoku geometry (c) Horse-Sudoku geometry

Figure 6.7: Domain decomposition for Sudoku problems.

The domain decomposition algorithm is completely transparent to the variant of the Sudoku problem since it only depends on the rules of the variant and the topology definition of the block (only in the case of jigsaw-Sudokus and Kakuros).

Kernel algorithm

The kernel code for solving the generic Sudoku problem is described as pseudocode on code 6.6. This section of the source code that is in charge of choosing the value that is not in the forbidden list is done by the master process in the `postGather` function (see pseudocode

6.7).

```
name  : Sudoku
input  : Graph representation of the domain received by the slave.
output: Graph representing the domain with new values of the problem.
foreach node ∈ partitionElements do
    if nodeEmpty then
        foreach neighbour ∈ neighbourhood do
            foreach setNumber ∈ neighbour do
                forbiddenNumbers ←+ setNumber
            end
        end
    end
end
```

Algorithm 6.6: Pseudocode of the Sudoku kernel algorithm.

```
name  : Sudoku postGather.
input  : Graph representation of the problem.
output: Graph representing the domain with new values of the problem.
foreach cell ∈ sudokuCells do
    if cellHasMultipleChoices then
        sortForbiddenValues ()
        removeDuplicated ()
        /* Searches for a value that is not in the forbidden numbers
           list.  If there is more than a value, the cell remains
           empty                                                    */
        searchValidValue ()
    end
end
```

Algorithm 6.7: Pseudocode of the postGather function.

6.6 Lattice-Boltzmann Method Test Application

LBM (Lattice Boltzmann Methods) is a class of CFD (Computational Fluid Dynamics) methods for fluid simulations. Instead of solving the Navier–Stokes equations, the discrete Boltzmann equation is solved to simulate the flow of a Newtonian fluid with collision models.

LBM is a simulation technique for complex fluid systems and has attracted interest from researchers in computational physics. Unlike the traditional CFD methods, which solve the conservation equations of macroscopic properties (ie. mass, momentum, and energy) numerically, LBM models the fluid consisting of fictitious particles, and such particles perform consecutive propagation and collision processes over a discrete lattice.

6.6.1 Methodology and parallelisation

LBM evolved out of LGCA (Lattice-Gas Cellular Automata) statistical models, inspired by the kinetic theory of gases.

LGCA and LBM are both sub-classes of CA (see explanation on page 115). Common characteristics for all of these models include:

- A set of connected points, the commonly called lattice.
- Some state-variables defined at each site. For the special case of a LGCA problem these variables are boolean while for the LBM are considered real variables.
- An update rule, based on the information of the local point and its neighbourhood. LBM and LGCA have composite update rules, commonly called *collision* and *streaming*.

The most important characteristic of the models is the discretisation of the velocity space, which means that the particle velocities are restricted to a finite set of orientations.

LBM consider particle distributions that live on the lattice nodes, rather than the individual particles, as the Navier-Stokes equations do. The general expression of the lattice Boltzmann equation is:

$$f_i(\vec{x} + \vec{e_i}\delta t, t + \delta t) = f_i(x, t) + \Omega_i \tag{6.10}$$

where the f_i is the concentration of particles that travels with velocity $\vec{e_i}$. With the discrete velocity $\vec{e_i}$ the particle distributions travel to the next lattice node in one time step δt. The collision operator Ω_i differs for the many lattice Boltzmann methods. In the lattice Boltzmann BGK (Bhatnagher-Gross-Krook) used [94], the particle distribution after propagation is

relaxed towards the equilibrium distribution $f_i^{eq}(x,t)$, as:

$$\Omega_i = \frac{1}{\tau}(f_i(x,t) - f_i^{eq}(x,t)) \tag{6.11}$$

where the relaxation τ parameter determines the viscosity of the simulated fluid.

The equilibrium distribution $f_i^{eq}(x,t)$ is a function of the local density ρ and the particle velocity $\vec{u}$:

$$f_i^{eq} = \omega_i \rho \left(1 + \frac{3\vec{e_i}\cdot\vec{u}}{c^2} + \frac{9(\vec{e_i}\cdot\vec{u})^2}{2c^4} - \frac{3\vec{u}^2}{2c^2}\right) \tag{6.12}$$

where the index i indicates de neighbour position for D2Q9-model as shown in the figure 6.8, c is the lattice velocity, ω_i in this case stands for:

$$\omega_i = \begin{cases} 4/9 & i = 0 \\ 1/9 & i = 1,2,3,4 \\ 1/36 & i = 5,6,7,8 \end{cases} \tag{6.13}$$

and:

$$e_i = \begin{cases} (0,0) & i = 0 \\ c(\cos(\frac{i-1}{2}\pi), \sin(\frac{i-1}{2}\pi)) & i = 1,2,3,4 \\ c(\cos(\frac{2i-1}{4}\pi), \sin(\frac{2i-1}{4}\pi)) & i = 5,6,7,8 \end{cases} \tag{6.14}$$

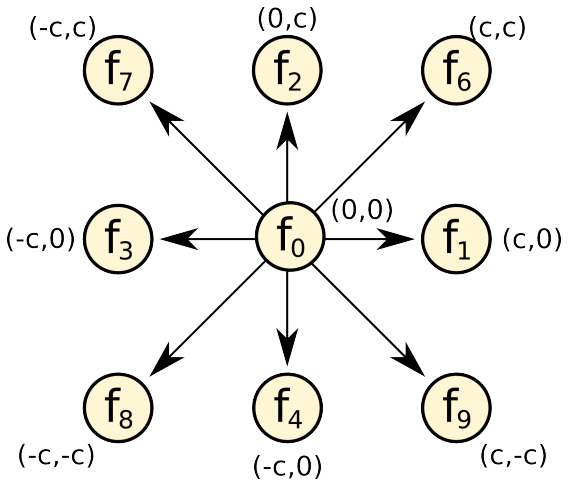


Figure 6.8: Matrices and e_i vectors for D2Q9-model.

The nomenclature of different models as D2Q9 or D3Q18 will be explained latter in subsection 6.6.2.

Due to its particular nature and local dynamics, LBM has several advantages over other conventional CFD methods, specially in dealing with complex boundaries, incorporating of microscopic interactions and the parallelisation of the algorithm.

6.6.2 Implementation

The LBM can be simulated by applying three phases: the *collide* phase, *stream* phase and the *interaction* phase (interaction with other objects rather than particles) [95].

In the first phase, the collision operator must be calculated. Here f_i are the discrete equilibrium particle probability distribution functions. The equations that define the collide phase, for a D2Q9-model, are obtained by joining the equations 6.12, 6.13 and 6.14. For a Moore neighbourhood (see image 6.2(b)) the expressions 6.15 are obtained.

$$\begin{aligned}
f_0(t+1) &= (1-\kappa)f_0(t) + \kappa\rho\omega_0\left(1-\tfrac{3}{2}|\vec{u}|^2\right)\\
f_1(t+1) &= (1-\kappa)f_0(t) + \kappa\rho\omega_1\left(1+3u_x+\tfrac{9}{2}u_x^2-\tfrac{3}{2}|\vec{u}|^2\right)\\
f_2(t+1) &= (1-\kappa)f_0(t) + \kappa\rho\omega_2\left(1-3u_x+\tfrac{9}{2}u_x^2-\tfrac{3}{2}|\vec{u}|^2\right)\\
f_3(t+1) &= (1-\kappa)f_0(t) + \kappa\rho\omega_3\left(1+3u_y+\tfrac{9}{2}u_y^2-\tfrac{3}{2}|\vec{u}|^2\right)\\
f_4(t+1) &= (1-\kappa)f_0(t) + \kappa\rho\omega_4\left(1-3u_y+\tfrac{9}{2}u_y^2-\tfrac{3}{2}|\vec{u}|^2\right)\\
f_5(t+1) &= (1-\kappa)f_0(t) + \kappa\rho\omega_5\left(1+3(u_x+u_y)+\tfrac{9}{2}(|\vec{u}|^2+u_{xy})-\tfrac{3}{2}|\vec{u}|^2\right)\\
f_6(t+1) &= (1-\kappa)f_0(t) + \kappa\rho\omega_6\left(1-3(u_x-u_y)+\tfrac{9}{2}(|\vec{u}|^2-u_{xy})-\tfrac{3}{2}|\vec{u}|^2\right)\\
f_7(t+1) &= (1-\kappa)f_0(t) + \kappa\rho\omega_7\left(1-3(u_x+u_y)+\tfrac{9}{2}(|\vec{u}|^2+u_{xy})-\tfrac{3}{2}|\vec{u}|^2\right)\\
f_8(t+1) &= (1-\kappa)f_0(t) + \kappa\rho\omega_8\left(1+3(u_x-u_y)+\tfrac{9}{2}(|\vec{u}|^2-u_{xy})-\tfrac{3}{2}|\vec{u}|^2\right)
\end{aligned} \tag{6.15}$$

In the expression shown before, ρ is the first moment of the distribution, $\rho = \sum f_i$; u_x is the velocity of the fluid in X-axis at time t:

$$u_x = f_1(t) - f_2(t) + f_5(t) - f_6(t) - f_7(t) + f_8(t) \tag{6.16}$$

u_y is the velocity of the fluid in Y-axis at time t:

$$u_y = f_3^i - f_4^i + f_5^i + f_6^i - f_7^i - f_8^i \tag{6.17}$$

u_{xy} is the cross coefficient $2u_xu_y$ and κ is the coupling factor (usually takes a value of 1).

In this phase, it is needed to apply the driven force to the particle probability distribution functions, as shown in expression 6.18.

$$
\begin{aligned}
f_0(t+1) &= f_0(t+1) - 2\omega_0\rho(F_{xx} + F_{yy}) \\
f_1(t+1) &= f_1(t+1) + \omega_1\,(3F_x + 6F_{xx} - 3F_{yy}) \\
f_2(t+1) &= f_3(t+1) + \omega_3\,(3F_y + 6F_{xx} - 3F_{yy}) \\
f_3(t+1) &= f_2(t+1) + \omega_2\,(-3F_x + 6F_{yy} - 3F_{yy}) \\
f_4(t+1) &= f_4(t+1) + \omega_4\,(-3F_y + 6F_{yy} - 3F_{yy}) \\
f_5(t+1) &= f_5(t+1) + \omega_5\,(3(F_x + F_y) + 6(F_{xx} + F_{yy}) + 9F_{xy}) \\
f_6(t+1) &= f_6(t+1) + \omega_6\,(-3(F_x - F_y) + 6(F_{xx} + F_{yy}) - 9FF_{xy}) \\
f_7(t+1) &= f_7(t+1) + \omega_7\,(-3(F_x + F_y) + 6(F_{xx} + F_{yy}) + 9F_{xy}) \\
f_8(t+1) &= f_8(t+1) + \omega_8\,(3(F_x - F_y) + 6(F_{xx} + F_{yy}) - 9F_{xy})
\end{aligned}
\tag{6.18}
$$

where $F_{xy} = F_x u_y + F_y u_x$, $F_{xx} = F_x u_x$ and $F_{yy} = F_y u_y$.

To calculate the initial values, the expression 6.15 may be used but, the values of the local velocities are the original ones. These local velocities are generally equal to the fluid velocity.

The stream phase consists of the movement of the object following the next rules:

$$
\begin{array}{lcll}
f_1(x,y,t) & \leftarrow & f_1(x-1,y,t); & \rightarrow\ W \\
f_2(x,y,t) & \leftarrow & f_2(x,y+1,t); & \rightarrow\ N \\
f_3(x,y,t) & \leftarrow & f_3(x+1,y,t); & \rightarrow\ E \\
f_4(x,y,t) & \leftarrow & f_4(x,y-1,t); & \rightarrow\ S \\
f_5(x,y,t) & \leftarrow & f_5(x-1,y+1,t); & \rightarrow\ NW \\
f_6(x,y,t) & \leftarrow & f_6(x+1,y+1,t); & \rightarrow\ NE \\
f_7(x,y,t) & \leftarrow & f_7(x+1,y-1,t); & \rightarrow\ SE \\
f_8(x,y,t) & \leftarrow & f_8(x-1,y-1,t); & \rightarrow\ SW
\end{array}
\tag{6.19}
$$

The third phase, *interaction with other objects*, changes the particles momentum of the particles that interact with objects in the fluid. In this case, the values of the implied matrices swap their values to each other minus the value that is related with the velocity of the object.

Data structure

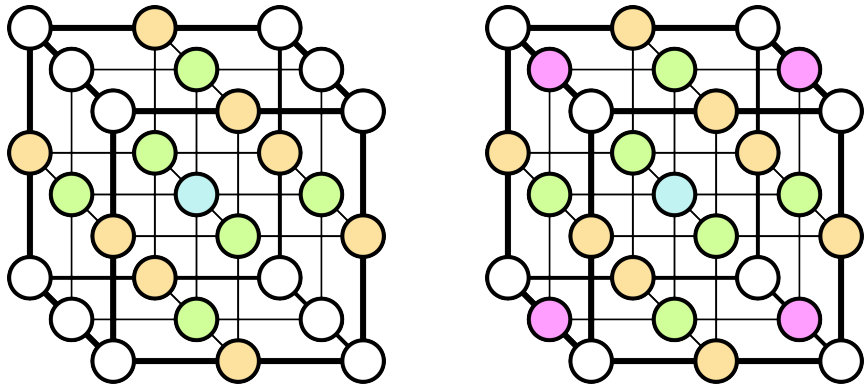
The user data structure that allows to integrate a sequential LBM simulation into the framework is as simple as a collection of the nine equilibrium distribution values, the velocities of the object and the external forces. This user object class can be shown at code 6.6.

```
20          public:
21              double f0, f1, f2, f3, f4, f5, f6, f7, f8, b;
22
23              Data();
24              Data(double);
25              virtual ~Data();
26              void initFData(double);
27              void print();
28
29              int size();
30
31              std::stringstream& operator<<(std::stringstream&);
32              std::stringstream& operator>>(std::stringstream&);
33      };
34  }
```

Code 6.6: Data structure definition for the LBM algorithm.

Neighbourhood

The lattice of LBM is usually classified using the $D\alpha Q\beta$ notation, where α is an integer number denoting the space dimensionality and β is the number of discrete directions at each lattice point (usually called velocity). Among the most commonly used lattices, the D2Q9-model (see figure 6.2(b)), the D3Q15-model and the D3Q19-model (see figure 6.9) can be found.



(a) D3Q15-Model. (b) D3Q19-Model.

Figure 6.9: Typical 3D LBM Lattices.

It is easy to notice that both expressions 6.15 and 6.19 can be simply parallelised thus, the only dependencies of each vertex of the graph can be represented as a Moore neighbourhood (see image 6.2(b)) lattice, where no interaction between two different domains can

happen.

Kernel algorithm

The code that simulates the LBM is described as pseudocode on 6.8, 6.9 and 6.10.

```
name : LBM
input : Graph representation of the domain received by the slave.
output: Graph representing the domain with new calculated values.
foreach node ∈ partitionElements do
    if step _is_even then
        Collide() (node)
        Stream() (node)
    else
        Collide() (node);
    end
end
```

Algorithm 6.8: Pseudocode of the LBM kernel algorithm applied to a Moore neighbourhood.

```
name : Collide phase
input : Graph representation of the domain received by the slave.
output: Graph representing the domain with new calculated values.
applyEquilibriumExpression (Node)
```

Algorithm 6.9: Pseudocode of the collide phase of the LBM.

```
name : Stream phase
input : Graph representation of the domain received by the slave.
output: Graph representing the domain with new calculated values.
applyMovementExpressions (Node)
```

Algorithm 6.10: Pseudocode of the stream phase of the LBM.

7

Results

In this chapter, the results achieved using the proposed platform are presented. These results are compared with the ones obtained by a single processor (sequential) in order to being able to visualise the performance obtained by applying the framework to the execution of the algorithms shown on the previous chapter.

The performance achieved by these algorithms has been measured with *Scalasca* [75] (refer to section 4.3), a software tool that supports the performance optimization of parallel programs by measuring and analysing their runtime behaviour. As explained before, this analysis is useful to identify potential performance bottlenecks - in particular those concerning communication and synchronisation procedures - offering guidance in exploring their causes to let the programmer to solve them. Due to an issue regarding the dynamic library loading procedure, it is not possible to profile the source code of the library as it is not known in profiling time and, therefore, there will be any measurement concerning the performance of the methods implemented at this level.

It is worth to remember that, the algorithms used as testbench are not the main goal of this work and they could be part of more dedicated researches. Hence, the purposes of the tests implemented are not to go into the specific physical issues and its meaning in depth, but to serve as performance testbench of the proposed platform.

As it has been made in the previous chapter, the algorithms are here expounded regarding its complexity, starting with the most simple one, the Left test application (refer to section 6.1) and finishing with the two most complex simulations, the LBM test application (refer to section 6.6) and the Sudoku solver test application (refer to section 6.5). For each of the algorithms, both the simulation results and the performance results are put forward.

7.1 Left Test Application

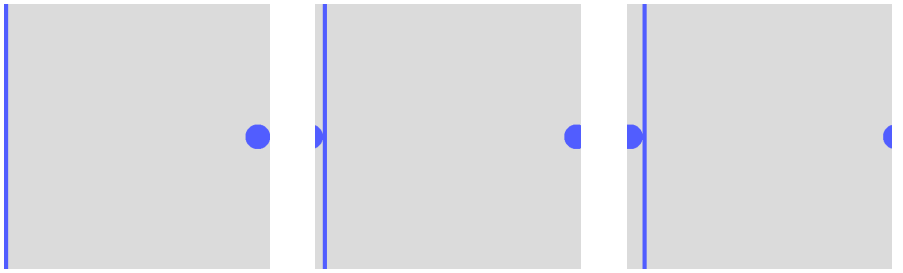
As it has been mentioned before, the Left test application has any research interest but it was the first implemented application and has been used as proof of concepts for all the features and improvements made to the framework. It is also the best example to explain the concepts and functionality of the platform, due to its simplicity.

This application can not scale and its parallelisation will worse the performance of the original and serial application, as it going to be shown through the results achieved by this algorithm, measured using the cited Scalasca profiler (see section 4.3).

The domain decomposition method chosen for this test application has been the BFS algorithm and it has not been yet tested with space filling methods.

7.1.1 Simulation results

The main goal of this application is not directly related to the concepts explained in this work, but to test the correctness of the platform, as simply as possible.



(a) Initial state

(b) Status of the simulation after 15 steps

(c) Status of the simulation after 30 steps

Figure 7.1: Three different steps of the Left algorithm executed within the platform. The size of the lattice is 1024×1024 px^2

The best way to test how correct is the framework, is by designing a very simple application that moves an element of a matrix (treated as an application graph) one step forward or backward. Once the application has been compiled and executed, it is very easy to graphically detect the computation errors, in case they exist. The input values used define a simple vertical line placed at the beginning left side of the lattice and a circle located at the right margin. Both elements must move forward (to the right side) one pixel per step (see figure 7.1). The line is used to verify that the pixels are moving, in this case, to the right side, whereas the circle demonstrate that the circle shape is not deformed while the execution, that is, all

cells of the lattice are store at the same position between different steps of the algorithm and therefore there is not a problem regarding the indices for accessing the graph that represents each partition from both points of view, the master execution and the slaves one.

If the framework spent more time in the internal platform procedures (graph creation and partition, load balancing policies, processes synchronisation and management, network operations, etc.) than in the application as itself (because of the simplicity of the serial code), the solution proposed to parallelise the serial application will not be the most accurate one and as result, the parallel implementation will worse the performance. In this case, it is said that the parallelisation does not make sense.

Processes		**Method** (s)					**Load**
	Total time	**main**	**Process**	**Kernel**	**Save**	**MPI_Probe**	**imbalance**(s)
2	0.870	0.082	0.011	0.313	0.001	0.413	0.169
3	2.110	0.239	0.012	0.721	0.001	0.905	0.732
5	4.299	0.609	0.012	0.766	0.001	2.579	1.193
9	8.271	0.673	0.011	0.808	0.001	5.124	1.243
17	12.790	0.708	0.014	0.887	0.001	10.525	1.325
33	11.338	0.403	0.012	1.077	0.001	8.661	0.626

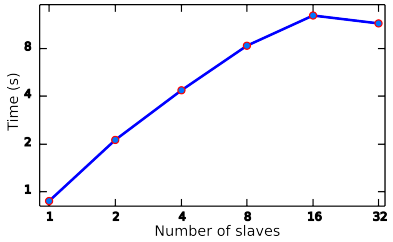
(a) Execution time in seconds vs. number of processes (the master process is also included). As it can be noticed, the sum of the execution times does not cover the total runtime. The remaining elapsed time is distributed along other primitives not shown in the table. The `main` method is executed by all the copies of the application and it instantiates the framework as a whole. The `Process` method reads the input values of the problem whereas the `Save` primitive execute the opposite action, it dumps the results into a formatted file. These two methods are only called by the master process, and therefore they are not affected by the number of processes and its time remains more or less constant during the testbench. The `Kernel` method is called only by the slave process and consists of the kernel algorithm source code. The `MPI_Probe` function waits until a message is received by the master process or the slave ones.

Processes		**Method** (%)				
	Total time	**main**	**Process**	**Kernel**	**Save**	**MPI_Probe**
2	0.870	9.19	0.10	35.63	0.02	47.13
3	2.110	11.31	0.10	34.15	0.06	42.88
5	4.299	14.16	0.27	17.83	0.03	60.00
9	8.271	8.10	0.12	9.79	0.01	61.91
17	12.790	5.54	0.11	6.93	0.01	82.29
33	11.338	3.55	0.02	9.50	0.01	76.39

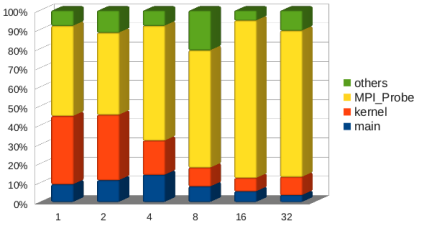
(b) Percentage of time spent in each method vs. number of processes (the master process also included). As it can be noticed, the total sum of the percentages is not 100%. The remaining time fraction is distributed along other primitives not shown in the table.

Table 7.1: Average execution time taken up by the platform and broken down into its most important methods. The application has been launched 30 times, each one 30 steps using a BFS domain decomposition algorithm. The size of the lattice is 1024×1024 px^2

The table 7.1(a) and figure 7.2(a) show the time behaviour of the application regarding the number of processes that join the simulation. Adding new processes to the simulation of the Left test application increase the global execution time, instead of reducing it. Hence, the



(a) Execution time running a parallel left test application. As explained above, the application, once parallelised is not scaling: adding new processes to the execution of the parallel algorithm increases the execution time instead of reducing it.



(b) Percentage of execution time spent on each of the most important methods of the parallel application. The bar "others" is used to achieve the 100% and it involves several methods out of the scope of this test. It can be observed that the application spent more time being blocked at the `MPI_Probe` primitive (function is in charge of sending and receiving the network packets) than in the execution of the test algorithm. The percentage of execution time spent on methods `main` and `Kernel` decreases because the execution time remains constant and the total time increases. The percentage of methods `Process` and `Save` are not shown, as their execution time can be neglected.

Figure 7.2: Execution time of the Left test application. The results were achieved by executing the algorithm within the platform over a 1024×1024 px^2 Cartesian lattice using as domain decomposition method, the BFS explained before. It can be observed that the behaviour of the runtime is similar to the one presented by the `MPI_Probe` primitive. This is due to the fact that the algorithm is very simple and all the behaviour of the system can be modelled only by the probe function.

parallel solution is not scaling, since the algorithm is simple as a memory values rotation.

The execution time has been split up into the main methods called by the framework. The ratio of time spent in each of these functions can be calculated by dividing the time spent in a certain method by the total execution time. These values can be seen in table 7.1(b) and in figure 7.2(b). The `main` method, if executed by the master process, is responsible for instantiating the framework that in turn reads the configuration files and the input data, creates both the application graph as the system graph, loads the load balancing policies and the plugins for the domain decomposition and leaves the platform prepared for the execution of the kernel method by the slave processes. The `Kernel` method is the one written by the end user of the application and executed only by the slave processes in parallel. As last, the `MPI_Probe` primitive (see description at page 239) blocks a process (master or slave) until it receives any network package. Hence, as biggest is the network package, as longest will be the process blocked at this primitive.

Load imbalance is a consequence of the `MPI_Probe` primitive, time spent on the creation of the network package, marshalling, transmission of packages, reception of them and unmarshalling procedure. As the time spent in the `Kernel` method decreases, there is less time to run the algorithm in parallel and therefore the total time needed to run the algorithm in full will be greater.

By comparing both tables and figures, it can be determined that in this case, the non-parallel behaviour of the algorithm running within the platform is completely a consequence of the `MPI_Probe` function.

Processes	**Imbalance** (s)	**Method** (%)			
		main	**Process**	**Kernel**	**Save**
2	0.17	13.34	0.18	77.55	0.47
3	0.73	21.78	0.40	65.60	0.23
5	1.19	64.32	1.59	25.71	0.18
9	1.24	70.37	1.77	17.46	0.18
17	1.32	78.25	1.92	8.73	0.19
33	0.63	58.87	0.74	13.83	0.41

Table 7.2: Average application performance taken up by the platform and broken down into its most important methods, vs. number of processes (the master process is also included). The application has been launched 30 times, each one 30 steps using a BFS domain decomposition algorithm. The size of the lattice is 1024×1024 px^2. As it can be noticed, the total sum of the percentages is not 100%. The remaining imbalance fraction is distributed along other primitives not shown in the table.

7.1.2 Performance results

As it has been shown before, the application will not scale and the parallelisation of this algorithm does not make sense from the standpoint to the performance, at least with the actual state of the platform. The values of the experiments can be shown in table 7.2.

In figure 7.3 is depicted how the load imbalance is divided up along the main methods of the application source code. The conclusion that we can obtain from this figure and its corresponding table, is that the performance of the application will be better as the total load imbalance decreases, but that is nearly the definition of the load imbalance. In case of having load imbalance, it is preferable to have it at the slave code, the one that runs on parallel.

The percentage of load imbalance of the `main` method grows with the number of processes of the simulation; this is due to the method used by the slaves to join the simulation, explained at the figure 5.5. The more processes join the execution of the parallel application, the more network packages must the master process wait for and, as the execution will not start until all the slaves have handshake the master, the more time must each slave wait until it can start with its execution. Hence, the slaves are waiting blocked for the master process, while their execution is as simple as a memory rotation, increasing the master process load imbalance. The load imbalance of the `Kernel` method (executed by the slave processes) must have the opposite behaviour: as more slave processes work together in the parallel application, as smallest are the network packages sent by the slaves to the master process, being less time blocked at the transmission procedure, reducing therefore the load imbalance of the slave processes execution, the `Kernel` procedure.

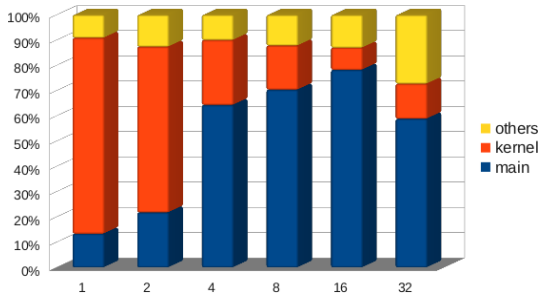


Figure 7.3: Distribution of the load imbalance along the main methods of the application. Most of the load imbalance happens in the `main` method, where the framework is instanced, the configuration files are read and the application and system graphs are created. As the number of processes grows, more time is spent on the handshaking phase, increasing the load imbalance of the `main` method, whereas the load imbalance of the `Kernel` method decreases, because the network packages are smaller and their transmission is fastest.

7.2 Game of Life Simulation Test Application

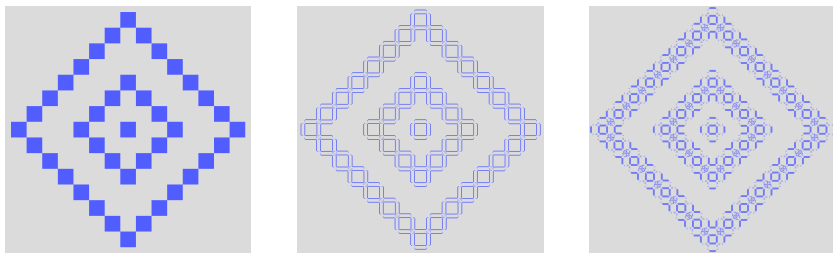
As shown before (refer to section 6.2), the GoL test application is a CA that simulates a scenario of live and dead cells and their evolution.

The rules to update the lattices are deterministic and simple enough to consider it as a mainly network massive test application. Thus, the domain decomposition method chosen for this test application has been, the BFS algorithm and it has not been yet tested with space filling methods.

7.2.1 Simulation results

For the set of experiments designed for testing this application within the framework, the HLRB cluster [96] (see characteristics at C) has been used. The neighbourhood is defined over a 2D Moore Euclidean 1024×1024 px^2 lattice.

The results achieved by the framework have been compared with those obtained by an external application, an application not related to the one implemented for this work. First of all, from the simulation point of view, making a comparison between the figures or images obtained by the external application and those obtained by the parallel algorithm, are completely equal. This demonstrates that the implementation of the framework is correct, covers the complete lattice of the problem, manages successfully the processes synchronisation and the thing that is more important, that the parallel techniques carried out by the platforms works perfectly with problems with the same characteristics of the GoL algorithm. Some images achieved by the platform can be shown on figure 7.4.



(a) Initial world status

(b) Status of the simulation after 10 steps

(c) Status of the simulation after 20 steps

Figure 7.4: Three different steps of the GoL algorithm executed within the platform. The size of the lattice is 1024×1024 px^2 and the neighbourhood of each alive or dead cell consists of a Moore Euclidean region (see figure 6.2(b)). Blue cells are alive whereas the whites are dead.

From the technical standpoint, the achieved results have been compared, in order to study how the integration with the platform affects the performance of the whole system. Not only is it important to know the total execution time but also split time taken up in each of the most important methods, by both the master and the slave processes (see table 148). It is worth to remember that the method `Process` is used by the master process for reading the initial values of the problem, while the `Save` one is just called by the master process for quite the opposite action, saving the results into a formatted file.

Processes		**Method** (s)					**Load**
	Total time	**main**	**Process**	**Kernel**	**Save**	**MPI_Probe**	**imbalance**(s)
2	23.776	0.785	0.013	2.150	0.001	18.838	1.466
3	10.944	0.536	0.002	1.826	0.001	6.673	1.024
5	4.510	0.426	0.002	1.496	0.001	2.001	1.121
9	3.647	0.409	0.002	1.353	0.001	1.455	1.310
17	4.284	0.758	0.013	1.190	0.001	2.091	1.828
33	7.906	0.485	0.002	1.832	0.001	4.614	1.269
65	63.009	1.267	0.010	2.466	0.001	52.673	2.019

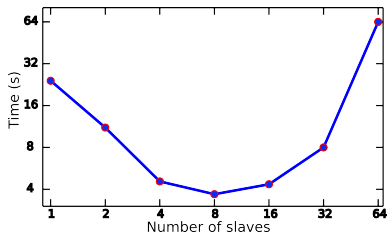
(a) Execution time in seconds vs. number of processes (the master process is also included). As it can be noticed, the sum of the execution times does not cover the total runtime. The remaining elapsed time is distributed along other primitives not shown in the table. The `main` method is executed by all the copies of the application and it instantiates the framework as a whole. The `Process` method reads the input values of the problem whereas the `Save` primitive execute the opposite action, it dumps the results into a formatted file. These two methods are only called by the master process, and therefore they are not affected by the number of processes and its time remains more or less constant during the testbench. The `Kernel` method is called only by the slave process and consists of the kernel algorithm source code. The `MPI_Probe` function waits until a message is received by the master process or the slave ones.

Processes		**Method** (%)				
	Total time	**main**	**Process**	**Kernel**	**Save**	**MPI_Probe**
2	23.776	3.30	0.01	9.04	0.00	79.23
3	10.944	4.89	0.02	16.69	0.01	60.97
5	4.510	9.44	0.03	33.18	0.02	44.37
9	3.647	11.20	0.04	13.38	0.02	39.90
17	4.284	17.69	0.29	27.77	0.02	48.88
33	7.906	6.13	0.02	24.35	0.01	58.35
65	63.009	2.01	0.02	3.91	0.00	83.48

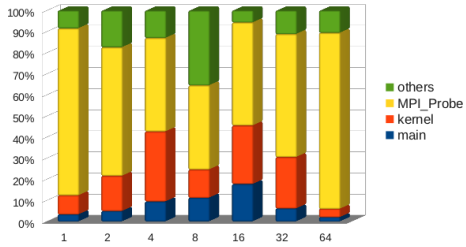
(b) Percentage of time spent in each method vs. number of processes (the master process is also included). As it can be noticed, the total sum of the percentages is not 100%. The remaining time fraction is distributed along other primitives not shown in the table.

Table 7.3: Average execution time taken up by the platform and broken down into its most important methods. The application has been launched 30 times, each one 30 steps using a BFS domain decomposition algorithm. The size of the lattice is 1024×1024 px^2 and the neighbourhood of each cell of the world consists of a Moore Euclidean region.

The table 7.3(a) and figure 7.5(a) show the time behaviour of the application regarding the number of processes that join the simulation. The execution time decreases adding new processes to the simulation until reaching a point, behaving as the expected scalability fea-



(a) Execution time running a parallel GoL test application. As explained above, the application, once parallelised scales until 9 processes (really 8 slave processes), a relative high value considering that the application is at most a network massive application.



(b) Percentage of execution time spent on each of the most important methods of the parallel application. The bar "others" is used to achieve the 100% and it involves several methods out of the scope of this test. It can be observed that the application spent more time being blocked at the `MPI_Probe` primitive (function is in charge of sending and receiving the network packets) than in the execution of the test algorithm. The percentage of execution time spent on the method `main` increases while the total time decreases, because the execution time remains constant as it is executed by the master process. Once the total time increases, the percentage of this method starts growing. The percentage of methods `Process` and `Save` are not shown, as their execution time can be neglected. The percentage of time used by the `kernel` method increases, as the performance of the application improves, meaning this, that the slave has more time to spent on the real algorithm proposed by the user and not on actions related to the platform. Once the percentage starts decreasing, the performance gets worse, except in the peak of performance, where the time spent on network actions has decreased until its smallest value.

Figure 7.5: Execution time of the GoL test application. The results were achieved by executing the algorithm within the platform over a 1024×1024 px^2 Moore lattice using as domain decomposition method, the BFS explained before. It can be observed that the behaviour of the runtime is similar to the one presented by the `MPI_Probe` primitive. This is due to the fact that the algorithm is very simple and all the behaviour of the system can be modelled only by the probe function.

ture. This point, in this test, is located between 9 and 17 running processes (really 8 and 16 slave processes). Upon this point, the execution time increases exponentially. The reason of this increase is due to the fact that the platform spends more time on the network procedures (information packaging/marshalling, buffer transmission, buffer reception and information unmarshalling) than in the computation of the kernel algorithm.

The execution time has been split up into the main methods called by the framework. The ratio of time spent in each of these functions can be calculated by dividing the time spent in a certain method by the total execution time. These values can be seen in table 7.3(b) and in figure 7.5(b). The `main` method, if executed by the master process, is responsible for instantiating the framework that in turn reads the configuration files and the input data, creates both the application graph as the system graph, loads the load balancing policies and the plugins for the domain decomposition and leaves the platform prepared for the execution of the kernel method by the slave processes. The `Kernel` method is the one written by the end user of the application and executed only by the slave processes in parallel. As last, the `MPI_Probe` primitive (see description at page 239) blocks a process (master or slave) until it receives any network package. Hence, as biggest is the network package, as longest will be the process blocked at this primitive.

Load imbalance is a consequence of the `MPI_Probe` primitive, time spent on the creation of the network package, marshalling, transmission of packages, reception of them and unmarshalling procedure. As the time spent in the `Kernel` method decreases, there is less time to run the algorithm in parallel and therefore the the total time needed to run the algorithm in full will be greater.

By comparing both tables and figures, it can be determined that in this case, the escalation behaviour of the algorithm running within the platform is completely a consequence of the `MPI_Probe` function, as both are affected in the same way.

7.2.2 Performance results

The more processes executes the algorithm, the more the problem is partitioned into domains and therefore, the more dependencies each partition has. As long as the domain is smallest, the partitions have more and more RO values, reducing the effectiveness of the decomposition method (see section 4.2.4) and increasing the load imbalance of the application. As in the case of the runtime shown before, the load imbalance decreases until a certain point, decreasing the global execution time. The values of the experiments can be shown in table 7.4.

In figure 7.6 is depicted how the load imbalance is divided up along the main methods of

Processes	Imbalance (s)	Method (%)			
		main	Process	Kernel	Save
2	1.47	56.80	1.77	20.03	0.13
3	1.02	50.88	0.45	23.32	0.18
5	1.12	30.10	0.29	54.20	0.14
9	1.31	21.26	0.19	68.83	0.10
17	1.83	28.98	0.71	65.10	0.05
33	1.27	35.80	0.26	45.13	0.13
65	2.02	67.84	0.99	12.44	0.10

Table 7.4: Average application performance taken up by the platform and broken down into its most important methods, vs. number of processes (the master process also included). The application has been launched 30 times, each one 30 steps using a BFS domain decomposition algorithm. The size of the lattice is 1024×1024 px^2 and the neighbourhood of each cell of the world consists of a Moore Euclidean region. As it can be noticed, the total sum of the percentages is not 100%. The remaining imbalance fraction is distributed along other primitives not shown in the table.

the application source code.

The percentage of load imbalance of the `main` method decreases with the number of processes of the simulation until a certain peak point which once reached, the load imbalance of the `main` method of the master execution increases. As it has been explained in the previous chapter, the load imbalance of the `main` method must increase with the number of processes, due to the handshake protocol (see figure 5.5) but, as more slaves join the execution and as in this case the slave algorithm consists of a more complex algorithm than in the previous test application, as more time is spent globally in the `Kernel` method, thwarting the effect of the handshake procedure. Once the peak point is reached, the communication and network transmission effect hides the execution carried out by the slave processes increasing the load imbalance of the `main` method.

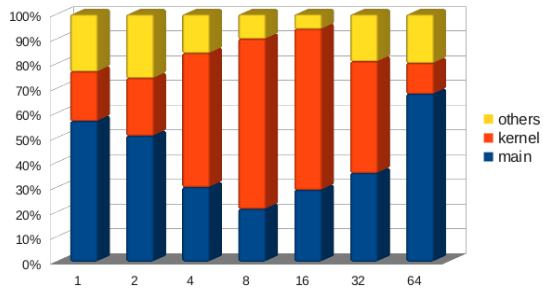


Figure 7.6: Distribution of the load imbalance along the main methods of the application. Most of the load imbalance happens in the `main` method, where the framework is instanced, the configuration files are read and the application and system graphs are created.

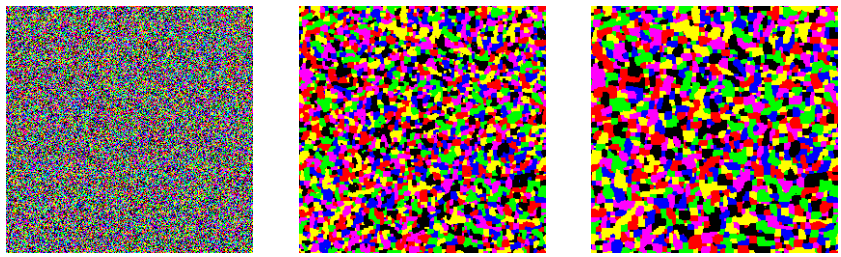
7.3 Potts simulation Test Application

The results of the parallel version have been compared with those obtained with the framework of the same problem. It is observed that the results obtained with this simulation are equivalent to those obtained with the standard Metropolis–Montecarlo method algorithm (non-parallelised one) [87]. This allows to assure that the domain decomposition and parallelisation technique has nothing to do with the Potts model and the Ferromagnetic properties from the point of view of the physical results.

Both, the problem GoL from the previous chapter and the Potts model are based on a CA that have a really very similar behaviour: while the game of live takes the values of its neighbourhood and applies a logical rule, the Potts model takes the value of its 4 neighbours and applies a simple statistical function that relates the values of the neighbourhood with a random value. Therefore, it is expected that the results of the Potts application are very similar to those obtained by GoL.

7.3.1 Simulation results

First of all, before starting to compare the execution times, as it has been done in previous chapters, it should be noted that the results obtained by running this parallel algorithm using the proposed platform are similar to those achieved using a similar serial algorithm, being therefore valid the tests done with this parallel Potts model. As an example, a sample execution is shown in figure 7.7.



(a) Initial appearance of the problem

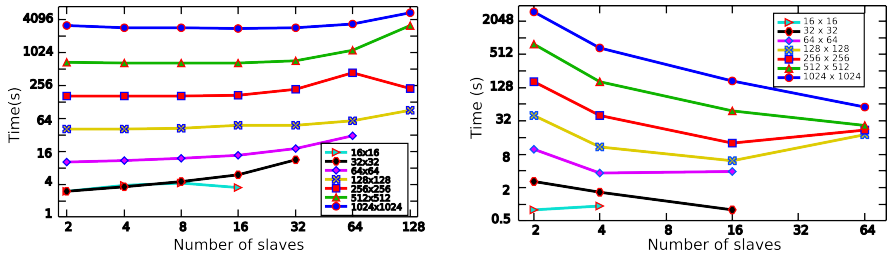
(b) Status of the simulation after 50 steps

(c) Status of the simulation after 100 steps

Figure 7.7: Three different steps of the Potts algorithm executed within the platform. The size of the lattice is 1024×1024 px^2 and the neighbourhood of each cell consists of a von Neumann Euclidean region (see figure 6.2(a)).

Five simulations have been performed to study the performance of the domain decomposition regarding this problem: one simulation was dedicated to the non-parallel applications

and four more to study the dependency and performance of the domain decomposition algorithm. In these simulations, the use of several domain decomposition were forced: based on horizontal and on vertical stripes, based on a square decomposition and the last one, using the framework explained before. These initial domain decompositions are needed to get the most performance values as possible, in order to study the behaviour of the framework depending on the initial input values, the initial distribution of nodes and the problem.



(a) Application execution time applying a stripes domain decomposition

(b) Application execution time applying a squares domain decomposition

Figure 7.8: Simulation results of the Potts test application applying the basic domain decomposition methods. The size of the lattice is 1024×1024 px^2 and the neighbourhood of each cell consists of a von Neumann Euclidean region.

Dividing the problem into stripes (refer to page 63 and figures 4.3, 4.4), the lattice is decomposed in vertical or horizontal domains, obtaining a global number N sub-domains (being N the number of slave processes). The main benefit of using this technique is the fact that it is a very simple algorithm because of the continuous memory architecture of computers. On the other hand, as disadvantage, this technique presents long border areas shared by each pair of neighbour processors; this implies a high probability of interaction between the 2 neighbour processors. Using this decomposition algorithm the execution time for simulating the problem will not scale. Time decreases adding, from the single processor version, a second processor to the simulation, but as we still adding new processes to the simulation, the execution time raises (see figure 7.8(a)). The reason of this behaviour resides on the great border area in comparison with the total area of the sub-domain received by the slaves. As the algorithm is computationally very simple, the more time is spent on creating the net packages than in solving the simulation.

Dividing the problem into squares (refer to page 63 and figure 4.5) is a very similar technique to stripes decomposition, but in this case, the sub-domain follows the geometry of the initial lattice and thus it takes into account the symmetries of the problem. The ratio of communications between pair of slave processors is negligible when the size of the lattice is much greater than zero ($d \gg 0$). The main drawback of this decomposition resides in the fact that the number of processes must be a square value. In this case, the execution time

presents a better performance of the system (see figure 7.8(b)).

Processes		Method (s)					Load
	Total time	main	Process	Kernel	Save	MPI_Probe	imbalance(s)
2	31.427	0.857	0.001	2.752	0.001	25.638	1.961
3	16,721	0.763	0.002	1.955	0.001	6.280	1.621
5	7.918	0.425	0.002	1.753	0.001	4.337	1.121
9	4.573	0.309	0.002	1.351	0.001	1.822	1.041
17	3.967	0.351	0.001	1.198	0.001	1.163	1.176
33	7.265	0.368	0.001	1.203	0.001	4.279	1.194
65	24.623	0.479	0.001	1.741	0.001	20.434	1.892
129	53.728	0.515	0.002	1.792	0.001	49.101	2.237

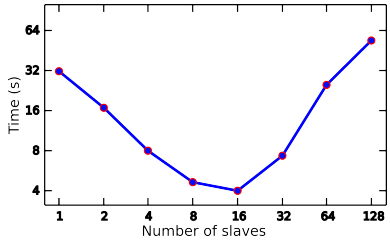
(a) Execution time in seconds vs. number of processes (the master process is also included). As it can be noticed, the sum of the execution times does not cover the total runtime. The remaining elapsed time is distributed along other primitives not shown in the table. The `main` method is executed by all the copies of the application and it instantiates the framework as a whole. The `Process` method reads the input values of the problem whereas the `Save` primitive execute the opposite action, it dumps the results into a formatted file. These two methods are only called by the master process, and therefore they are not affected by the number of processes and its time remains more or less constant during the testbench. The `Kernel` method is called only by the slave process and consists of the kernel algorithm source code. The `MPI_Probe` function waits until a message is received by the master process or the slave ones.

Processes		Method (%)				
	Total time	main	Process	Kernel	Save	MPI_Probe
2	31.427	2.73	0.01	8.76	0.00	81.58
3	16.721	4.56	0.01	11.69	0.01	37.56
5	7.918	5.37	0.03	22.14	0.01	54.77
9	4.573	6.76	0.04	29.54	0.02	39.84
17	3.967	8.85	0.03	30.20	0.03	29.32
33	7.265	5.07	0.01	16.56	0.01	58.90
65	24.623	1.95	0.01	7.07	0.01	82.99
129	53.728	0.96	0.01	3.34	0.01	91.39

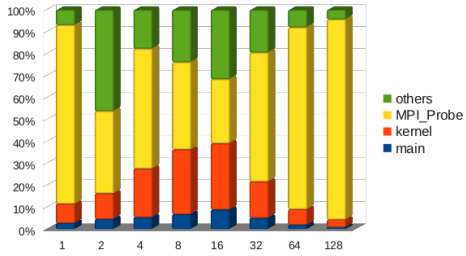
(b) Percentage of time spent in each method vs. number of processes (the master process is also included). As it can be noticed, the total sum of the percentages is not 100%. The remaining time fraction is distributed along other primitives not shown in the table.

Table 7.5: Average execution time taken up by the platform and broken down into its most important methods. The application has been launched 30 times, each one 30 steps using a Hilbert space filling domain decomposition algorithm. The size of the lattice is 1024×1024 px^2 and the neighbourhood of each cell of the world consists of a von Neumann Euclidean region.

The table 7.5(a) and figure 7.9(a) show the time behaviour of the application regarding the number of processes that join the simulation. The execution time decreases adding new processes to the simulation until reaching a point, behaving as the expected scalability feature. This point, in this test, is located at 17 (really 16 slave processes) running processes. Upon this point, the execution time increases exponentially. The reason of this increase is due to the fact that the platform spends more time on the network procedures (information packaging/marshalling, buffer transmission, buffer reception and information unmarshalling) than in the computation of the kernel algorithm.



(a) Execution time running a parallel Potts test application. As explained above, the application, once parallelised scales until 17 processes (really 16 slave processes).



(b) Percentage of execution time spent on each of the most important methods of the parallel application. The bar "others" is used to achieve the 100% and it involves several methods out of the scope of this test. It can be observed that the application spent more time being blocked at the `MPI_Probe` primitive (function is in charge of sending and receiving the network packets) than in the execution of the test algorithm. The percentage of execution time spent on the method `main` increases while the total time decreases, because the execution time remains constant as it is executed by the master process. Once the total time increases, the percentage of this method starts growing. The percentage of methods `Process` and `Save` are not shown, as their execution time can be neglected. The percentage of time used by the `kernel` method increases, as the performance of the application improves, meaning this, that the slave has more time to spent on the real algorithm proposed by the user and not on actions related to the platform. Once the percentage starts decreasing, the performance gets worse, except in the peak of performance, where the time spent on network actions has decreased until its smallest value.

Figure 7.9: Execution time of the Potts test application. The results were achieved by executing the algorithm within the platform over a 1024×1024 px^2 von Neumann lattice using as domain decomposition method, the Hilberts curve explained before.

The execution time has been split up into the main methods called by the framework. The ratio of time spent in each of these functions can be calculated by dividing the time spent in a certain method by the total execution time. These values can be seen in table 7.5(b) and in figure 7.9(b). The `main` method, if executed by the master process, is responsible for instantiating the framework that in turn reads the configuration files and the input data, creates both the application graph as the system graph, loads the load balancing policies and the plugins for the domain decomposition and leaves the platform prepared for the execution of the kernel method by the slave processes. The `Kernel` method is the one written by the end user of the application and executed only by the slave processes in parallel. As last, the `MPI_Probe` primitive (see description at page 239) blocks a process (master or slave) until it receives any network package. Hence, as biggest is the network package, as longest will be the process blocked at this primitive.

Load imbalance is a consequence of the `MPI_Probe` primitive, time spent on the creation of the network package, marshalling, transmission of packages, reception of them and unmarshalling procedure. As the time spent in the `Kernel` method decreases, there is less time to run the algorithm in parallel and therefore the the total time needed to run the algorithm in full will be greater.

The application, once parallelised scales until 17 processes (really 16 slave processes), a relative high value considering that the application is at most a network massive application.

7.3.2 Performance results

Processes	**Imbalance** (s)	**Method** (%)			
		main	**Process**	**Kernel**	**Save**
2	1.96	61.02	0.63	33.74	0.13
3	1.62	51.33	0.55	42.97	0.18
5	1.12	42.71	0.46	53.77	0.14
9	1.04	28.76	0.32	68.33	0.10
17	1.18	30.24	0.37	62.23	0.05
33	1.19	31.37	0.40	59.72	0.13
65	1.89	33.68	0.43	57.33	0.10
129	2.24	40.11	0.45	47.21	0.12

Table 7.6: Average application performance taken up by the platform and broken down into its most important methods, vs. number of processes (the master process also included). The application has been launched 30 times, each one 30 steps using a Hilbert space-filling curve as domain decomposition algorithm. The size of the lattice is 1024×1024 px^2 and the neighbourhood of each cell of the world consists of a von Neumann Euclidean region. As it can be noticed, the total sum of the percentages is not 100%. The remaining imbalance fraction is distributed along other primitives not shown in the table.

In figure 7.10 is depicted how the load imbalance is divided up along the main methods of the application source code. The conclusion that we can obtain from this figure and its corresponding table, is that the performance of the application will be better as the load imbalance decreases, but that is nearly the definition of the load imbalance. In case of having load imbalance, it is preferable to have it at the slave code, the one that runs on parallel.

The percentage of load imbalance of the `main` method decreases with the number of processes of the simulation until a certain peak point which once reached, the load imbalance of the `main` method of the master execution increases. As it has been explained in the previous chapter, the load imbalance of the `main` method must increase with the number of processes, due to the handshake protocol (see figure 5.5) but, as more slaves join the execution and as in this case the slave algorithm consists of a more complex algorithm than in the previous test application, as more time is spent globally in the `Kernel` method, thwarting the effect of the handshake procedure. Once the peak point is reached, the communication and network transmission effect hides the execution carried out by the slave processes increasing the load imbalance of the `main` method.

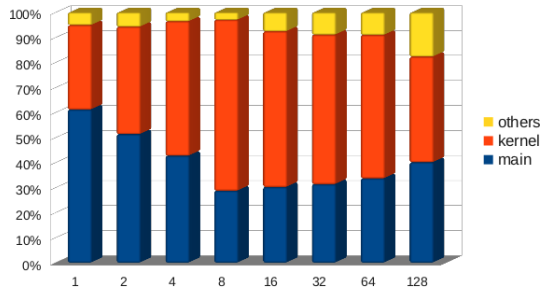


Figure 7.10: Distribution of the load imbalance along the main methods of the application. Most of the load imbalance happens in the `main` method, where the framework is instanced, the configuration files are read and the application and system graphs are created.

7.4 NEP simulator Test Application

The NEP application was initially developed to test the platform with the `string` management classes and with something different of an application based on a mathematical model. Coincidentally, some people of a research group of the University of the author had developed a platform for NEPs and they wanted to run that platform in a parallel way. The first idea was launched using a Java Party thread implementation, but the result was not as accurate as expected. Because of this, it was decided to test that application with the environment that has been proposed here.

The main feature that distinguishes this test application from the other ones is the fact that the application graph of the previous applications had a constant set of values, that is, the cardinality of the set of `Data` values between two different steps of the simulation was identical, there were any new value but, in the case of the NEP test application, this is completely different: the nodes of the application graphs generate new values in each step of the simulation, growing therefore the set of `Data` values of the graph (see figure 7.11). This is very important from the standpoint of the behaviour of the platform, because as the algorithm progresses, the set of values generated grows exponentially, turning from an application more or less intense on computing into a traffic network massive application. This application is therefore categorized as a mixed application located between the massive computation and massive network. It is worth to remember that the set of values for each vertex of the application graph must be serialised and sent to the master process for gathering the results.

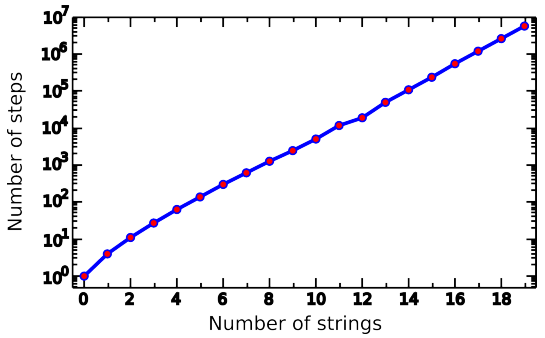


Figure 7.11: Number of generated strings that reach the final NEP element per step.

The results achieved have been taken with a 32-NEP.

7.4.1 Simulation results

First of all, to validate the application and to be able to compare the behaviour of the parallel application regarding different scenarios, the obtained results must be reviewed and manually checked for possible erroneous results. In this case as in the previous test application, the results are similar to the ones that could be obtained by an equivalent sequential application. As the results obtained by the application consist of thousand and thousand of strings, it is not worth to include them here.

Two sets of experiments have been performed: one on a conventional multi-core architecture and one on a massively parallel platform.

For the first set of experiments we have used a multi-threaded multi-core platform (a desktop computer running a Linux kernel 2.6.26, with 16Gb of memory and $4 \star 6$ cores Intel(R) Xeon(R) CPU E7450 2.40GHz) running a Java multi-threaded simulator for NEPs, developed by our research group. The jNEP (Java Networks of Evolutionary Processors) platform was able to solve graphs up to 8 nodes whereas the biggest graph solved by our parallel framework had 24 nodes.

The results, for the second set, have been obtained by executing a sequential NEP kernel in a parallel environment (HLRB (Hochleichtungrechnenzentrum Bayern) II, 9728 cores, 4Gb memory per core [96]) using the described framework. The simulation has been executed with NEPs of several number of nodes, from $n = 16$ (more or less 4×10^3 valid strings) up to $n = 24$ (5×10^5 strings). From $n = 28$ and higher values, the assigned resources reached the limit. To observe the framework behaviour the number of slaves has been changed, from 2^0 (equivalent to single processor) to 2^4. It is not possible to have 2^5 or more slaves, because this exceeds the number of vertex of the NEP. That is the reason for our limited testbench.

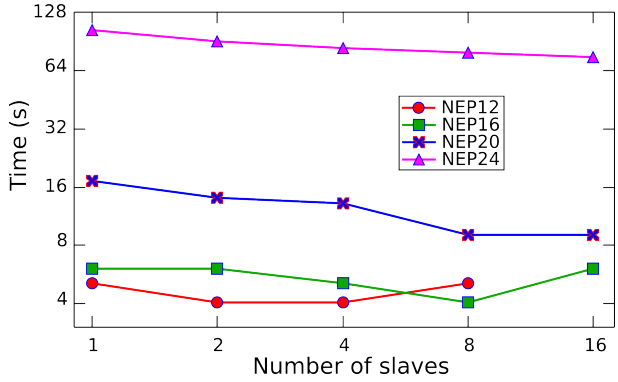


Figure 7.12: Execution time versus number of running processes. The performance of the application has improved running the NEP algorithm in comparison with the single slave execution.

From the point of view of the execution time, we can observe that the performance of the algorithm has not been worsened by the use of the framework (see figure 7.12). It can be also observed that the execution time decreases until a certain value, that depends on the number of processors and on the dimension of the problem, is reached. Once this point is exceeded, if the number of processors still grows, the execution time will start growing again, just because the master spends more time on the management of the communication, processes and domains than the slaves on the real calculus of the problem.

Processes		**Method** (s)					**Load**
	Total time	**main**	**Process**	**Kernel**	**Save**	**MPI_Probe**	**imbalance**(s)
2	103.372	0.873	0.001	21.728	6.241	68.834	3.027
3	90.144	0.721	0.001	17.526	6.328	63.276	1.971
5	83.418	0.756	0.001	16.443	6.114	58.349	1.584
9	79.721	0.674	0.001	14.782	6.237	56.371	1.392
17	74.289	0.623	0.001	12.782	6.179	53.865	1.269

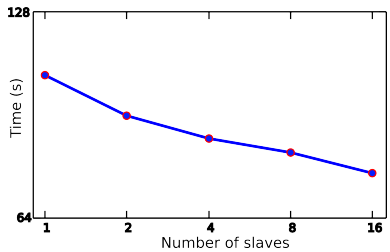
(a) Execution time in seconds vs. number of processes (the master process is also included). As it can be noticed, the sum of the execution times does not cover the total runtime. The remaining elapsed time is distributed along other primitives not shown in the table. The `main` method is executed by all the copies of the application and it instantiates the framework as a whole. The `Process` method reads the input values of the problem whereas the `Save` primitive execute the opposite action, it dumps the results into a formatted file. These two methods are only called by the master process, and therefore they are not affected by the number of processes and its time remains more or less constant during the testbench. The `Kernel` method is called only by the slave process and consists of the kernel algorithm source code. The `MPI_Probe` function waits until a message is received by the master process or the slave ones.

Processes		**Method** (%)				
	Total time	**main**	**Process**	**Kernel**	**Save**	**MPI_Probe**
2	103.372	0.01	0.00	21.02	6.04	66.59
3	90.144	0.01	0.00	19.44	7.02	70.19
5	83.418	0.01	0.00	19.71	7.33	69.95
9	79.721	0.01	0.00	18.54	7.82	70.71
17	74.289	0.01	0.00	16.44	8.32	72.51

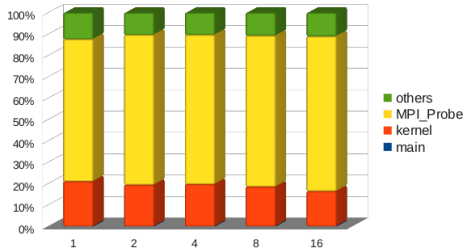
(b) Percentage of time spent in each method vs. number of processes (the master process is also included). As it can be noticed, the total sum of the percentages is not 100%. The remaining time fraction is distributed along other primitives not shown in the table.

Table 7.7: Average execution time taken up by the platform and broken down into its most important methods. Load imbalance is a consequence of the `MPI_Probe` primitive (see description at page 239), time spent on the creation of the network package, marshalling, transmission of packages, reception of them and unmarshalling procedure.

The table 7.7(a) and figure 7.13(a) show the time behaviour of the application regarding the number of processes that join the simulation. The execution time decreases adding new processes to the simulation but it is not behaving as the expected scalability feature. Th reason of this new behaviour comes from the time needed to send and save all the string generated by the slave processes. It is worth to highlight the fact that the number of strings is growing exponentially.



(a) Execution time running a parallel NEP test application. As explained above, the application, once parallelised scales until 9 processes, a relative high value considering that the application is at most a network massive application.



(b) Percentage of execution time spent on each of the most important methods of the parallel application. The bar "others" is used to achieve the 100% and it involves several methods out of the scope of this test. It can be observed that the application spent more time being blocked at the `MPI_Probe` primitive (function is in charge of sending and receiving the network packets) than in the execution of the test algorithm. The percentage of the `Process`, `postGather` and `Save` methods does not appear at this plot, as they are completely meaningful with respect of the time spent by the other methods.

Figure 7.13: Execution time of the NEP test application.

The execution time has been split up into the main methods called by the framework. The ratio of time spent in each of these functions can be calculated by dividing the time spent in a certain method by the total execution time. These values can be seen in table 7.7(b) and in figure 7.13(b). The `main` method, if executed by the master process, is responsible for instantiating the framework that in turn reads the configuration files and the input data, creates both the application graph as the system graph, loads the load balancing policies and the plugins for the domain decomposition and leaves the platform prepared for the execution of the kernel method by the slave processes. The `Kernel` method is the one written by the end user of the application and executed only by the slave processes in parallel. As last, the `MPI_Probe` primitive (see description at page 239) blocks a process (master or slave) until it receives any network package. Hence, as biggest is the network package, as longest will be the process blocked at this primitive.

Load imbalance is a consequence of the `MPI_Probe` primitive, time spent on the creation of the network package, marshalling, transmission of packages, reception of them and unmarshalling procedure. As the time spent in the `Kernel` method decreases, there is less time to run the algorithm in parallel and therefore the the total time needed to run the algorithm in full will be greater.

Processes	Imbalance (s)	Method (%)			
		main	Process	Kernel	Save
2	3.03	58.76	0.82	39.23	0.19
3	1.97	51.34	0.65	46.71	0.19
5	1.58	49.32	0.53	48.01	0.18
9	1.39	47.58	0.51	51.13	0.17
17	1.27	44.91	0.48	53.48	0.07

Table 7.8: Average application performance taken up by the platform and broken down into its most important methods, vs. number of processes (the master process also included). As it can be noticed, the total sum of the percentages is not 100%. The remaining imbalance fraction is distributed along other primitives not shown in the table.

7.4.2 Performance results

In figure 7.8 is depicted how the load imbalance is divided up along the main methods of the application source code. The conclusion that we can obtain from this figure and its corresponding table, is that the performance of the application will be better as the load imbalance decreases, but that is nearly the definition of the load imbalance. In case of having load imbalance, it is preferable to have it at the slave code, the one that runs on parallel.

The percentage of load imbalance of the `main` method decreases with the number of processes of the simulation in a very smoothy way. As it has been explained in the previous chaptera, the load imbalance of the `main` method must increase with the number of processes, due to the handshake protocol (see figure 5.5) but, as more slaves join the execution and as in this case the slave algorithm consists of a more complex algorithm than in the previous test application, as more time is spent globally in the `Kernel` method, thwarting the effect of the handshake procedure. Furthermore, the effect of the `Save` method (executed by the master process) can even hide the effect of the computations made by the slaves at the `Kernel` method, balancing both of them. This issue, explains why both imbalances are so similar.

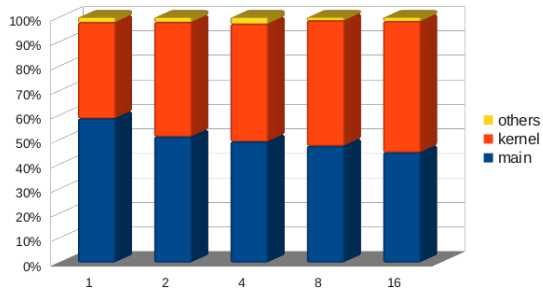


Figure 7.14: Distribution of the load imbalance along the main methods of the application.

7.5 Sudoku solver Test Application

The main feature that distinguishes this Sudoku solver test application from the other implemented applications is the fact that the Sudoku solver presents an heterogeneous communication pattern, being this fact the reason of the implementation of this algorithm within the proposed platform. In other applications, the neighbourhood of each of the cells that make up the grid did not depend on the position it occupies in the grid (except for the cells in the borders without boundary conditions), but simply on the closest neighbours defined by the type of grid (square, triangular or hexagonal) (see figure 1.3). In the case of the Sudoku solver, the neighbourhood consists of two sets, one with those fixed cells for each position (the cells of the row and the column) and another one that varies depending on the position, the cell inside the block. Depending on where the cell is located on the block, its neighbours change their position relative to the current cell. This feature has not been tested yet with the other applications.

Each cell defines 24 dependencies (each cell affects 24 more cells), defining thereby a total of 81×24 data dependencies in a 9×9 tiles Sudoku. This causes the problem to be strongly coupled, and since the algorithm used is simple, the application can be categorized as an application of massive network traffic instead as a compute intensive one.

7.5.1 Simulation results

First, as it has been done with the other applications, before analysing the results achieved by running the application within the framework, it should be checked that the results are the ones expected and even more in this case, where a new functionality of the platform, which can make the calculations to be completely wrong, is being tested.

The Sudoku configuration (initial distribution of the numbers) used as test is simple

		2		1	9			4
		3	8	2	4	5		1
		1	7	6		3	2	
9						6	8	3
				4				
2	1	5						7
	6	9		8	3	2		
1		4	5	7	6	8		
5			1	9		4		

(a) Status of the problem after 1 step

		2	3	1	9	7		4
		3	8	2	4	5		1
		1	7	6	5	3	2	
9		7	2	5		6	8	3
				4				
2	1	5		3	8	9		7
7	6	9	4	8	3	2		5
1		4	5	7	6	8		9
5			1	9	2	4		6

(b) Status of the problem after 2 steps

8		2	3	1	9	7	6	4
6		3	8	2	4	5	9	1
4	9	1	7	6	5	3	2	8
9	4	7	2	5	1	6	8	3
		6	9	4	7	1	5	2
2	1	5		3	8	9	4	7
7	6	9	4	8	3	2	1	5
1	2	4	5	7	6	8	3	9
5	3	8	1	9	2	4	7	6

(c) Status of the problem after 4 steps

Figure 7.15: Three different steps of the Sudoku solver algorithm executed within the platform. The size of the lattice is 9×9 tiles. The neighbourhood of each cell consists of an heterogeneous region as shown in figure 6.5(a).

enough to being able to solve it manually in order to validate the algorithm and therefore giving sense to the measures shown in this section.

The actual implementation of the Sudoku algorithm allows to only solve puzzles of 9×9 cells[2]. The problem is so small that a priori the application will no scale properly. The problem also resides on the strings C++ class implementation that it is not the optimal one. Furthermore, the application has been implemented as a fairly similar distributed agents application, where each agent sends its cell number to all of the cells that form its neighbourhood, giving a little more sense to the implementation of this application.

The table 7.9(a) and figure 7.16(a) show the time behaviour of the application regarding the number of processes that join the simulation. As shown in both references, the execution time increases by adding a new third process (second slave) to the execution. This is related to the application is tightly coupled. Adding few slaves to the simulation does not implies improving the application performance, since the problem is still tightly coupled and besides of having to solve the problem, it must deal with the management of new processes. Adding new processes can break this situation, as it happens in this case. The execution time decreases adding new processes to the simulation until reaching a point, located in this example at nearly 9 (really 8 slave processes) running processes. This optimal value can be expected, as there are 9 distinct domains per 9×9 tiles sudoku (9 rows or 9 columns or 9 blocks). Upon this point, the execution time increases exponentially. The reason of this increase is due to the fact that the platform spends more time on the management of the processes than in the computation of the kernel algorithm.

The execution time has been split up into the main methods called by the framework. The ratio of time spent in each of these functions can be calculated by dividing the time spent in a certain method by the total execution time. These values can be seen in table 7.9(b)

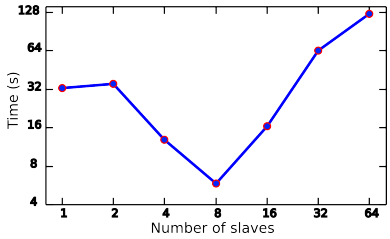
Processes		Method (s)						Load
	Total time	main	Process	Kernel	Save	pG	MPI_Probe	imbalance(s)
2	32.117	0.567	0.001	9.781	0.003	0.005	12.454	9.007
3	34.965	0.986	0.001	16.134	0.017	0.003	12.133	4.926
5	12.739	0.389	0.001	4.154	0.002	0.003	3.392	3.411
9	5.829	0.235	0.001	1.580	0.001	0.002	1.549	1.857
17	16.289	0.584	0.001	6.029	0.017	0.003	6.039	2.089
33	63.215	1.274	0.001	30.667	0.023	0.005	10.117	21.086
65	122.633	1.652	0.001	56.528	0.047	0.003	17.309	46.681

(a) Execution time in seconds vs. number of processes (the master process is also included). As it can be noticed, the sum of the execution times does not cover the total runtime. The function name *pG* stands for the `postGather` method. The remaining elapsed time is distributed along other primitives not shown in the table. The `main` method is executed by all the copies of the application and it instantiates the framework as a whole. The `Process` method reads the input values of the problem whereas the `Save` primitive execute the opposite action, it dumps the results into a formatted file. These two methods are only called by the master process, and therefore they are not affected by the number of processes and its time remains more or less constant during the testbench. The `Kernel` method is called only by the slave process and consists of the kernel algorithm source code. The `MPI_Probe` function waits until a message is received by the master process or the slave ones.

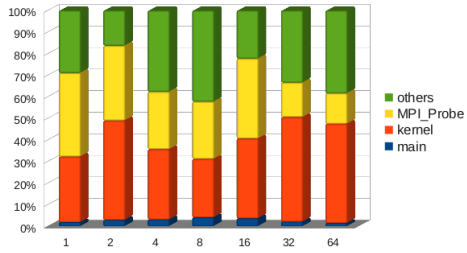
Processes		Method (%)					
	Total time	main	Process	Kernel	Save	pG	MPI_Probe
2	32.177	1.77	0.00	30.45	0.01	0.02	38.78
3	34.965	2.82	0.00	46.14	0.05	0.01	34.70
5	12.739	3.05	0.01	32.61	0.02	0.02	26.63
9	5.829	4.03	0.02	27.11	0.02	0.03	26.57
17	16.289	3.59	0.01	37.01	0.10	0.02	37.07
33	63.215	2.02	0.00	48.51	0.04	0.01	16.00
65	122.633	1.35	0.00	46.10	0.04	0.00	14.11

(b) Percentage of time spent in each method vs. number of processes (the master process is also included). As it can be noticed, the total sum of the percentages is not 100%. The remaining time fraction is distributed along other primitives not shown in the table.

Table 7.9: Average execution time taken up by the platform and broken down into its most important methods. Load imbalance is a consequence of the `MPI_Probe` primitive (see description at page 239), time spent on the creation of the network package, marshalling, transmission of packages, reception of them and unmarshalling procedure.



(a) Execution time running a parallel Sudoku solver test application. As explained above, the application, once parallelised scales until 9 processes, a relative high value considering that the application is at most a network massive application.



(b) Percentage of execution time spent on each of the most important methods of the parallel application. The bar "others" is used to achieve the 100% and it involves several methods out of the scope of this test. It can be observed that the application spent more time being blocked at the `MPI_Probe` primitive (function is in charge of sending and receiving the network packets) than in the execution of the test algorithm. The percentage of execution time spent on the method `main` increases while the total time decreases, because the execution time remains constant as it is executed by the master process. Once the total time increases, the percentage of this method starts growing. The percentage of the `Process`, `postGather` and `Save` methods does not appear at this plot, as they are completely meaningful with respect of the time spent by the other methods. The percentage of time used by the `kernel` method increases, as the performance of the application improves, meaning this, that the slave has more time to spent on the real algorithm proposed by the user and not on actions related to the platform. Once the percentage starts decreasing, the performance gets worse, except in the peak of performance, where the time spent on network actions has decreased until its smallest value.

Figure 7.16: Execution time of the Sudoku solver test application.

and in figure 7.16(b). The `main` method, if executed by the master process, is responsible for instantiating the framework that in turn reads the configuration files and the input data, creates both the application graph as the system graph, loads the load balancing policies and the plugins for the domain decomposition and leaves the platform prepared for the execution of the kernel method by the slave processes. The `Kernel` method is the one written by the end user of the application and executed only by the slave processes in parallel. As last, the `MPI_Probe` primitive (see description at page 239) blocks a process (master or slave) until it receives any network package. Hence, as biggest is the network package, as longest will be the process blocked at this primitive. A new method has been profiled, the `postGather` primitive. This function

Load imbalance is a consequence of the `MPI_Probe` primitive, time spent on the creation of the network package, marshalling, transmission of packages, reception of them and unmarshalling procedure. As the time spent in the `Kernel` method decreases, there is less time to run the algorithm in parallel and therefore the the total time needed to run the algorithm in full will be greater.

7.5.2 Performance results

Processes	**Imbalance** (s)	**Method** (%)				
		main	**Process**	**Kernel**	**Save**	**pG**
2	9.01	45.27	0.65	48.17	0.06	0.15
3	4.93	49.78	0.77	38.32	0.07	0.15
5	3.41	34.49	0.43	57.33	0.06	0.11
9	1.86	22.63	0.21	72.10	0.07	0.13
17	2.09	44.74	0.38	51.55	0.14	0.17
33	21.09	63.28	0.51	28.14	0.10	0.15
65	46.86	71.43	0.82	21.84	0.09	0.16

Table 7.10: Average application performance taken up by the platform and broken down into its most important methods, vs. number of processes (the master process also included). As it can be noticed, the total sum of the percentages is not 100%. The remaining imbalance fraction is distributed along other primitives not shown in the table. The function name *pG* stands for the `postGather` method.

In figure 7.17 is depicted how the load imbalance is divided up along the main methods of the application source code. The conclusion that we can obtain from this figure and its corresponding table, is that the performance of the application will be better as the load imbalance decreases, but that is nearly the definition of the load imbalance. In case of having load imbalance, it is preferable to have it at the slave code, the one that runs on parallel.

The percentage of load imbalance of the `main` method decreases with the number of

processes of the simulation in a very smoothly way. As it has been explained in the previous chapter, the load imbalance of the `main` method must increase with the number of processes, due to the handshake protocol (see figure 5.5) but, as more slaves join the execution and as in this case the slave algorithm consists of a more complex algorithm than in the previous test application, as more time is spent globally in the `Kernel` method, thwarting the effect of the handshake procedure. Furthermore, the effect of the `Save` method (executed by the master process) can even hide the effect of the computations made by the slaves at the `Kernel` method, balancing both of them. This issue, explains why both imbalances are so similar.

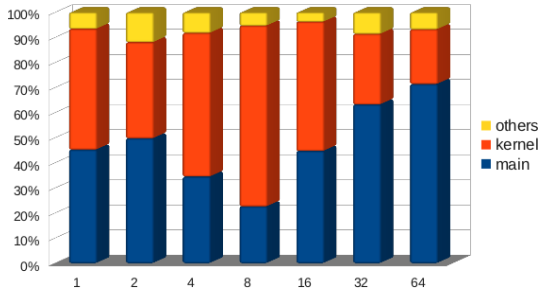


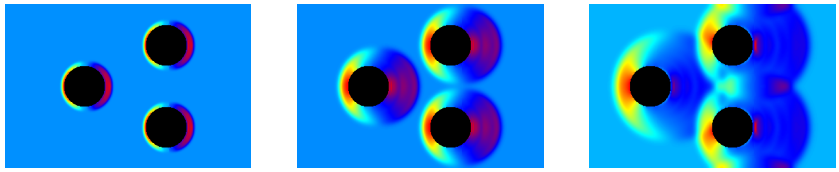
Figure 7.17: Distribution of the load imbalance along the main methods of the application.

7.6 Lattice-Boltzmann Method Test Application

This last application used as testbench for the proposed framework is the most computational intensive from the set of experiments designed for this work. The early first idea was given by the people of the University of Erlangen-Nürnberg, where the author made her first research foreign stay, but at that moment, the platform was extreme difficult to use and, instead of developing the LBM method, it was decided to rewrite the complete framework. With the new framework (the actual one) the LBM method can be integrated.

7.6.1 Simulation results

The results achieved by the framework have been compared with those obtained by an external application, an application not related to the one implemented for this work. First of all, from the simulation point of view, making a comparison between the figures or images obtained by the external application and those obtained by the parallel algorithm, are completely equal. Some images achieved by the platform can be shown on figure 7.18.



(a) Status of the problem after 10 step

(b) Status of the problem after 50 steps

(c) Status of the problem after 90 steps

Figure 7.18: Three different steps of the LBM algorithm executed within the platform. The size of the lattice is 1024×1024 px^2 and the neighbourhood of each alive or dead cell consists of a Moore Euclidean region (see figure 6.2(b)).

From the technical standpoint, the achieved results have been compared, in order to study how the integration with the platform affects the performance of the whole system. Not only is it important to know the total execution time but also split time taken up in each of the most important methods, by both the master and the slave processes (see table 170). It is worth to remember that the method `Process` is used by the master process for reading the initial values of the problem, while the `Save` one is just called by the master process for quite the opposite action, saving the results into a formatted file. The `postGather` method permits to execute an operator to the global domain since the slaves have not the information to execute it, partially, each over its received domain. This method is therefore executed only by the master process.

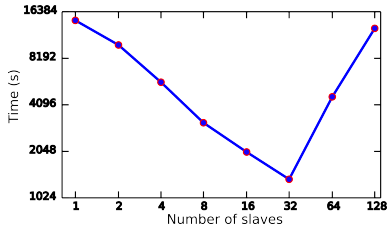
Processes		Method (s)						Load
	Total time	main	Process	Kernel	Save	pG	MPI_Probe	imbalance(s)
2	14400	728.32	1.462	2930	576.16	12.00	8932	5880
3	9894	734.92	1.470	1874	570.19	12.02	6543	3734
5	5723	740.69	1.460	971	581.23	12.01	3141	2138
9	3128	735.67	1.462	450	560.89	12.02	1067	1714
17	2021	731.47	1.458	197	549.19	12.00	509	921
33	1343	720.01	1.471	27	560.74	12.03	19	856
65	4582	735.94	1.461	1384	565.33	12.02	1772	1834
129	12700	739.79	1.458	2778	563.55	12.03	7310	5615

(a) Execution time in seconds vs. number of processes (the master process is also included). As it can be noticed, the sum of the execution times does not cover the total runtime. The remaining elapsed time is distributed along other primitives not shown in the table. The `main` method is executed by all the copies of the application and it instantiates the framework as a whole. The `Process` method reads the input values of the problem whereas the `Save` primitive execute the opposite action, it dumps the results into a formatted file. These two methods are only called by the master process, and therefore they are not affected by the number of processes and its time remains more or less constant during the testbench. The `Kernel` method is called only by the slave process and consists of the kernel algorithm source code. The `MPI_Probe` function waits until a message is received by the master process or the slave ones.

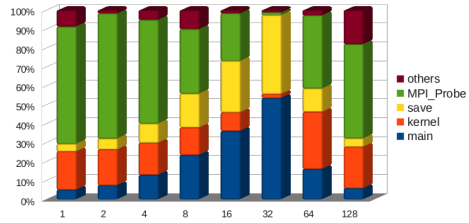
Processes		Method (%)					
	Total time	main	Process	Kernel	Save	pG	MPI_Probe
2	14400	5.05	0.01	20.35	3.94	0.08	62.03
3	9894	7.43	0.01	18.95	5.76	0.12	66.14
5	5723	12.94	0.02	16.97	10.16	0.21	54.89
9	3128	23.53	0.04	14.41	17.93	0.38	34.14
17	2021	36.19	0.07	9.79	27.17	0.59	25.22
33	1343	53.61	0.11	2.06	41.75	0.89	1.43
65	4582	16.06	0.03	30.21	12.34	0.26	38.69
129	12700	5.84	0.01	21.95	4.44	0.09	49.84

(b) Percentage of time spent in each method vs. number of processes (the master process is also included). As it can be noticed, the total sum of the percentages is not 100%. The remaining time fraction is distributed along other primitives not shown in the table.

Table 7.11: Average execution time taken up by the platform and broken down into its most important methods. The function name *pG* stands for the `postGather` method. The application has been configured to run 100 steps using a BFS domain decomposition algorithm. The size of the lattice is 1024×1024 px^2 and the neighbourhood of each cell of the world consists of a Moore Euclidean region.



(a) Execution time running a parallel LBM test application. As explained above, the application, once parallelised scales until 33 processes (really 32 slave processes), a relative high value considering that the application is at most on a network operation.



(b) Percentage of execution time spent on each of the most important methods of the parallel application. The bar "others" is used to achieve the 100% and it involves several methods out of the scope of this test. It can be observed that the application spent more time being blocked at the `MPI_Probe` primitive (function is in charge of sending and receiving the network packets) than in the execution of the test algorithm. The percentage of execution time spent on the method `main` increases while the total time decreases, because the execution time remains constant as it is executed by the master process. Once the total time increases, the percentage of this method starts growing. The percentage of the `Process` and `postGather` methods does not appear at this plot, as they are completely meaningful with respect of the time spent by the other methods. The percentage of time used by the `kernel` method increases, as the performance of the application improves, meaning this, that the slave has more time to spent on the real algorithm proposed by the user and not on actions related to the platform. Once the percentage starts decreasing, the performance gets worse, except in the peak of performance, where the time spent on network actions has decreased until its smallest value. The `Save` method behaves completely in the opposite way, as the time consumption is close to be considered as constant.

Figure 7.19: Execution time of the LBM test application. The results were achieved by executing the algorithm within the platform over a 1024×1024 px^2 Moore lattice using as domain decomposition method, the BFS explained before.

The table 7.11(a) and figure 7.19(a) show the time behaviour of the application regarding the number of processes that join the simulation. The execution time has been split up into the main methods called by the framework. The ratio of time spent in each of these functions can be calculated by dividing the time spent in a certain method by the total execution time. These values can be seen in table 7.11(b) and in figure 7.19(b). The `main` method, if executed by the master process, is responsible for instantiating the framework that in turn reads the configuration files and the input data, creates both the application graph as the system graph, loads the load balancing policies and the plugins for the domain decomposition and leaves the platform prepared for the execution of the kernel method by the slave processes. On the other hand, the method `postGather` applies an operator to the global domain of the problem, as the slaves have not the complete information to apply this operator to its domain. The `Kernel` method is the one written by the end user of the application and executed only by the slave processes in parallel. As last, the `MPI_Probe` primitive (see description at page 239) blocks a process (master or slave) until it receives any network package. Hence, as biggest is the network package, as longest will be the process blocked at this primitive.

As it can be deduced from the image 7.19(a) and the table 7.11(a), the execution time decreases adding new processes to the simulation until reaching a point, behaving as the expected scalability feature. This point, in this test, is located at the amount of 33 running processes (really 32 slave processes). Upon this point, the execution time increases exponentially. The reason of this increase is due to the fact that the platform spends more time on the network procedures (information packaging/marshalling, buffer transmission, buffer reception and information unmarshalling) than in the computation of the kernel algorithm, increasing the time spent in the `MPI_Probe` primitive.

Load imbalance is a consequence of the `MPI_Probe` primitive, time spent on the creation of the network package, marshalling, transmission of packages, reception of them and unmarshalling procedure. As the time spent in the `Kernel` method decreases, there is less time to run the algorithm in parallel and therefore the the total time needed to run the algorithm in full will be greater.

7.6.2 Performance results

The more processes executes the algorithm, the more the problem is partitioned into domains and therefore, the more dependencies each partition has. As long as the domain is smallest, the partitions have more and more RO values, reducing the effectiveness of the decomposition method (see section 4.2.4) and increasing the load imbalance of the application. As in the case of the runtime shown before, the load imbalance decreases until a certain point, decreasing the global execution time. The values of the experiments can be shown in

table 7.12.

Processes	**Imbalance** (s)	**Method** (%)				
		main	**Process**	**Kernel**	**Save**	**pG**
2	5888	5.71	0.03	33.17	58.13	0.27
3	3734	4.93	0.02	35.71	56.37	0.20
5	2138	4.20	0.02	40.54	52.73	0.18
9	1714	3.75	0.01	42.84	50.98	0.15
17	921	2.01	0.02	45.89	48.39	0.13
33	856	1.76	0.01	49.36	46.76	0.17
65	1834	1.95	0.02	50.55	45.22	0.18
129	5615	2.07	0.03	49.48	45.65	0.21

Table 7.12: Average application performance taken up by the platform and broken down into its most important methods, vs. number of processes (the master process also included). The application has run 100 steps using a BFS domain decomposition algorithm. The size of the lattice is 1024×1024 px^2 and the neighbourhood of each cell of the world consists of a Moore Euclidean region. As it can be noticed, the total sum of the percentages is not 100%. The remaining imbalance fraction is distributed along other primitives not shown in the table. The function name *pG* stands for the `postGather` method.

In figure 7.20 is depicted how the load imbalance is divided up along the main methods of the application source code.

The percentage of load imbalance of the `main` method decreases with the number of processes of the simulation until a certain peak point which once reached, the load imbalance of the `main` method of the master execution increases. As it has been explained in the previous chapter, the load imbalance of the `main` method must increase with the number of processes, due to the handshake protocol (see figure 5.5) but, as more slaves join the execution and as in this case the slave algorithm consists of a more complex algorithm than in the previous test application, as more time is spent globally in the `Kernel` method, thwarting the effect of the handshake procedure. Once the peak point is reached, the communication and network transmission effect hides the execution carried out by the slave processes increasing the load imbalance of the `main` method.

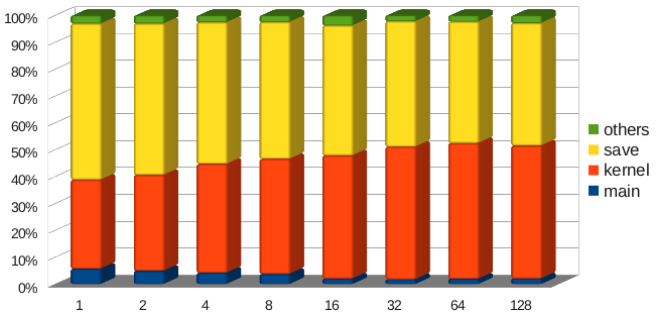


Figure 7.20: Distribution of the load imbalance along the main methods of the application. Most of the load imbalance happens in the `main` method, where the framework is instanced, the configuration files are read and the application and system graphs are created.

7.7 Analysis of the results

Comparing the results obtained by running all the test applications, the following conclusions can be drawn:

- It makes no sense to parallelise those applications whose execution time is smaller than the time spent by the management of the framework, understanding management as the processes management, network traffic, and serialisation and deserialisation of objects, network frames encapsulation, management of the load balancing policies and the partitioning of the domain. In these cases, the application performance will not scale. The best example for this is the old British saying “too many hands spoil the broth”.
- Massive communication applications have a poorer performance and a worst scalability compared with massive computation applications. That is completely trivial, as the platform acts or tries to optimise the parallelisation of the algorithms but it has nothing to do related to the network communications.
- Except the Left test application, the remaining ones scale, in a greater or lesser way, from the point of view of the execution time and therefore of the performance one.
- Comparing the percentage of the tables of time spent per method, the applications scale while the percentage of the `main` method decrease, increasing the time of the `kernel` method, method that actually performs the parallelisation, since it is invoked from the slave processes.
- The application that scale, do it up to a point at which the runtime explodes. Once that point is reached, the load imbalance of the application starts growing. This issue can be easily explained applying the first point seen in this section: reaching this “special

point" means spending more time on internal procedures of the framework than in the proper `kernel` algorithm.

- The time spent on methods `process`, `save` and `postGather` (when it exists) remains constant and independent of the number of processes that join the execution. As those methods are only executed by the master process, they are not affected by the slave procedures. The imbalance of these methods does not behave in the same way, as the global load imbalance is distributed along the methods, independently from the process that execute them and, proportional to the global time spent on the simulation.
- The load imbalance is a directly consequence of the `MPI_Probe` function, as it is mostly related to the communication and transmission actions. This function is the main bottleneck of all the applications shown in this chapter.

III

Conclusions and Future Work

8

Conclusions

The platform balances the workload properly using the load balancing plugins specified by the user and it decomposes, according to the method set up by the end user, the domain correctly as it can be observed by running test applications proposed and explained. It is also observed that, regardless of the problem simulated using the proposed platform, the simulation results are the correct ones, there is no missing information between steps of the algorithm and it there are not erroneous values detected in the intermediate results obtained. The tests have also determined that the user interaction, despite being tedious, is correct and allows the definition of data, algorithms and configurations needed for the proper working of the sequential algorithm within the platform.

From the point of view of the platform performance, it must be highlight that the performance of the serial algorithm has not been worsened by the use of the framework, except, as expected, in the case of the first application explained, the *Left* application. This can not be considered as a failure of the platform as it has been explained in the results section of that application (refer to chapter 9).

The platform, as it has been explained over the chapters presented here, even when it is a version with some errors that must be corrected in future versions, it is perfectly usable and meets the goals initially proposed.

However, not all the conclusions will be praise for the platform. It still has some bugs and items that should be improved to enhance user interaction. As an example, the Sudoku application has only been tested with standard 9×9 Sudokus, because the configuration file to define the lattice for that problem has already more than 2000 lines, which have to be written by the user without the option to generate them graphically or using an automated external program or even an internal component.

As it has been demonstrated in the results section, the platform still takes too long in simple tasks performed by the master process, as for example, the creation of the application graph or the tasks that involve network transactions.

As shown by the results obtained, the network is the major bottleneck of the application, making inevitable the fact of having to optimise all the actions that have something to do with the transfer of information between processes. This topic could extensively improve the performance of the application which, in turn, led more importance to the development of this platform.

9 Future Work

It is said that in a work such as the presented here, development is never truly finished, it is only halted. This is certainly true in this case; there are just many things to test, to polish, to add, to... But at some moment a line must be drawn. In the case of this work, it has been drawn at the point in which the platform presented had begun to be effective from the standpoint of the functionality and features and when working with it was not place an additional burden on a user familiar with concepts here exposed.

Since the author was well aware of the changes that would need to be made, design has been kept as modular as possible, issue that can be seen through the use of plugins, abstract classes ..., allowing an easy replacement of almost any part of the process and also making easier the implementation and integration of any different behaviour or improvement of the platform.

As a final point in this work we describe those elements that, being uncompleted for the current state of this documentation because they were considered as secondary or even further away from the main line, share a direct relationship with the researched ones. The research lines and enhancements proposed in this version would require a large investment of time an effort to be tested and refined. Even though only some of them will be actively pursued, they are all worth mentioning. These points can be categorized into the following schema:

- Reduce the limitations of the platform.
- Add new features.
- New testbenchs.

These three points will be explained on the following sections.

9.1 Reduce the limitations of the platform

As it has been explained in section 1.4.1 the platform still has several limitations that may face small puzzles to users of this platform.

It is mainly about the fact of the memory failure, where not all the memory that is has been reserved is freed, hindering the users to run a larger number of steps of the same algorithm although this depends obviously on the amount of memory than the algorithm needs. Furthermore, related to the same issue, it rarely happens a *core dump–segmentation fault* using the classic pointers instead of using the C++ standard containers. These two faults are widely linked and it is possible that it is a single failure showing two different symptoms. The solution to this limitation is simple but it involves a minimal change of mentality of the user of the platform while implementing the algorithms.

From another complete different point of view, there is a relevant limitation that may affect the global performance of the platform; the dynamical creation and management of processes. The integration of this method within the platform, consists on the use of a standard primitive of the MPI-2 library, `MPI_Comm_spawn()` which its mail goal is the creation of new processes on the fly after a MPI application has been started.

MPI has nowadays several popular versions: version 1.3 (shortly called MPI-1), which emphasized message passing and had a static runtime environment, and version 2.2 (commonly named MPI-2), which includes new features such as parallel I/O, dynamic process management and remote memory operations.

The MPI-2 specification describes three main interfaces by which MPI processes can dynamically establish communications, `MPI_Comm_spawn()`, `MPI_Comm_accept()` (via the primitive `MPI_Comm_connect()`) and `MPI_Comm_join()`. The `MPI_Comm_spawn()` interface allows an MPI process to spawn a number of instances of the named MPI process. The newly spawned set of MPI processes form a new `MPI_COMM_WORLD` (MPI multicast) intracommunicator (see definition communicator) but it can communicate with the parent process using the intercommunicator returned by the function.

This last improvement of the platform is maybe de most complex and important one but, although it sounds very critical, not all the problems will have the same behaviour once this solution is implemented, some will improve their performance but some other may also reduce them. Furthermore, as the software will became more complex from the computational standpoint, it will be expected to initially have more runtime errors during the execution of the new solution.

9.2 Add new features

The platform presented here fulfils several basic functionalities that allow to execute sequential code in a parallel way though, to continue developing this tool, it is intended to improve the following:

9.2.1 Partitioning algorithm

Regarding the partitioning algorithm, the following improvements can be found:

Functional decomposition

This work, regarding the partition computational work feature, is focused on domain decompositions, leaving the functional decomposition to a future work.

It is important to remember that the domain decomposition techniques are those that part the problem in groups (domains) and applies the same set of instructions (algorithm) to each partition, whereas the techniques of functional decomposition breaks a program into tasks which are applied on the same data set.

Functional decomposition lends itself well to problems that can be easily split into different tasks.

Differentiation between “figure” and “ground”

The standards domain decomposition algorithms included in the framework decompose the complete lattice into several parts, without attending whether the information of that domain is “part of the figure” or “part of the background” (see image 9.1), terms also called “positive” and “negative” spaces.

Why should the background be partitioned if there is no relevant information there? And further more, why should the figure be equally decomposed if there is more information in some places than in others? The platform implemented, by default, does not consider this option as there is no way to define the meta-information needed by the user to specify “which information is the one useful for the algorithm”. The only possibility is to do this differentiation manually in the domain decomposition plugin but this option does not prune the graph for erasing the unnecessary vertices; at least, if new areas of information appears after the execution of any step of the parallelised algorithm, as the platforms works with forests (refer to the definition of tree), the new values may be taken into account for next steps, thus

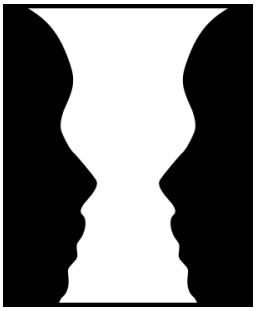
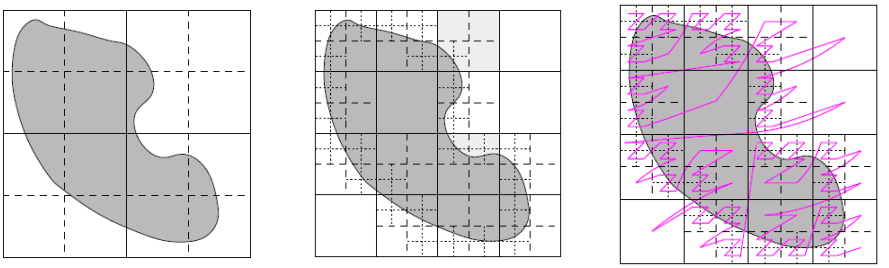


Figure 9.1: What's here depicted? Two black faces over a white background or a white vase in a black background?

facilitating the implementation of this improvement to the partitioning and therefore the overall system performance.

Regarding the second questions placed before, there are recursive methods [97] that take care of this, such us for example quadtree for 2D problems [98] (see figure 9.2) or octree for those defined on 3D spaces. The behaviour is similar to an image segmentation or the community detection on graphs [99, 100].



(a) Homogeneous quadtree decomposition. Both the "figure" and the "background" are decomposed with the same level of details

(b) Heterogeneous quadtree decomposition. The "figure" is decomposed applying more quadrants where there is more information. The "background" is nearly ignored

(c) Heterogeneous quadtree decomposition following a space filling curve. The "figure" is decomposed applying more quadrants where there is more information. The "background" is nearly ignored

Figure 9.2: Two different applications of the quadtree algorithm over a generic surface. The first one decomposes the figure equally since the second one applies a more detailed grain where the figure is located

New research tendencies are mixing these recursive methods with space filling curves (see figure 9.2(c) [101–103]).

Code optimisations

The use of the cited library, *Metis* [55], for decomposing the application graph into domains could accelerate the master procedures, probably reducing the load imbalance of this process and improving the global performance of the platform.

On the other side, the C++ BGL (Boost Graph Library) provides a generic interface to create, access and manage graph structures, hiding the details of the implementation. This is an "open" interface in the sense that any graph library that implements this interface will be interoperable with the BGL generic algorithms and with other algorithms that also use this interface. The BGL graph interface and graph components are generic, in the same sense as the STL (Standard Template Library) [104]. The use of this library could optimise and simplify the master and slaves procedures.

9.2.2 Fault-tolerance and error recovering

Failures are irremediable; they just happen. Any software development is error free. Even in the best scenario, with ideal conditions, the framework can fail, processes can get out of memory and the packets reception can be delayed until the framework dies from starvation. This is inherent to computing science and the use of exceptions is not enough to prevent the framework from failing as long as the error conditions have been checked and evaluated before the errors take place.

Since it is not possible to prevent a program from errors, the important things are the MTBF (Mean Time Between Failures) (available description at MTBF) and the minimisation of the consequences of failures. How? It must be assured the continue proper execution of the platform after the event of one or more failures within one or more components of the system. This is the so often called fault-tolerant property. Fault-tolerant forces isolation techniques to the failing component and containment to prevent the propagation of the failure to other components. From the point of view of the isolation of the failing component, the platform adopts the decision to send packets only to those nodes that have on time and correctly answered to a previous request. It is considered that the messages that have not been received from a particular process, will arrive the master process with a timestamp equal to infinity and therefore, applying the expressions to calculate the number of tasks to a certain process (see expressions 4.1 and 4.2) will be 0 tasks for that process. Although the platform fulfils the first of the requirements, it can not be said that the platform is totally fault-tolerant, as the second requirement, "containment to prevent the propagation of the failure to other components", is not obeyed in the present version.

The concept "error recovering" (also known as "recover from errors") is somehow related

to the fault-tolerant property, as it is about the procedures to recover the missing or erroneous information after a failure has happened. Error recovering tries to answer questions as the ones planed on this situation: “suppose a cluster formed by several nodes where the simulation of a certain algorithm is taking place. The lattice of the problem is decomposed into domains that are sent to each of the slave processes. The slaves apply the algorithm to the received domain to obtain the values of the next step of the algorithm. Once each process obtains these values, it sends them back to the master using the network interface configured for that issue. One of the slaves is unable to reach the master process and thus, it can not send the obtained results to the master process. The master process gathers the results of all the slaves except the ones of the slave that is having problems with the network interface. At this point, the master can not wait until the process sends the results but, those results must be done by the slaves, that are still alive from the point of view of the communication with the master, in order to proceed with the normal execution of the algorithm”. The policies to determine the behaviour of the framework in such situations is part of the error recovering procedure. The actual state of the platform does not take into account this situations and the error recovering is something that has been planed as future work and improvements of the proposed solution.

9.2.3 Improve the performance of the saving procedure

As it has been already mentioned several times in this document, the I/O and the communication operations are the two actions, being non–parallel procedures, that worsen the performance of the parallel application (in our case, the performance of platform). Minimise the communications between processes is part of partitioning algorithm and therefore is a responsibility of the end-user of the system but, with respect to the I/O operations minimisation, the user has little to do.

I/O is a major bottleneck in many parallel applications. The main reason for poor application-level I/O performance is that parallel-I/O systems are optimized for large accesses (on the order of megabytes), whereas parallel applications typically make many small I/O requests (on the order of kilobytes or even less).

One way to improve the performance of the application is by improving the performance of the saving procedure executed by the master process using the corresponding plugin written by the user (see page 13). This I/O operation could be done in parallel taking advantage of the MPI–I/O primitives.

A key feature of MPI–IO is that it allows users to access several non-contiguous pieces of data from a file with a single I/O function call by defining file views with derived datatypes. The parallel I/O feature refers to a set of functions designed to abstract I/O management on

distributed systems, and allow files to be easily accessed using the existing derived datatype functionality.

Datatypes in MPI are of two kinds: basic and derived. Basic datatypes are those that correspond to the basic datatypes in the host programming language. In addition, MPI provides datatype-constructor functions to create derived datatypes consisting of multiple basic datatypes located either contiguously or non–contiguously.

This improvement has not jet implemented because of the little research that has been done on this feature and the difficulty to get a good performance [105]. For example, some implementations of sparse matrix-vector multiplications using the MPI–I/O library are disastrously inefficient [106].

9.2.4 Optimise the master procedures

Related with the previous section, there is also the possible improvement of the procedures that run at the master side. The master process execution consists also of a serial code that has not yet been parallelised.

The procedure that takes more time within the master execution is the code of creating the graphs (the application and the system graphs) from the values and configurations written by the user in the correspondent files. The complexity on time grows as $O(N \times M)$ where N stands for the number of vertices of the graph and M for the cardinality of the biggest set of links of the vertices of the graph.

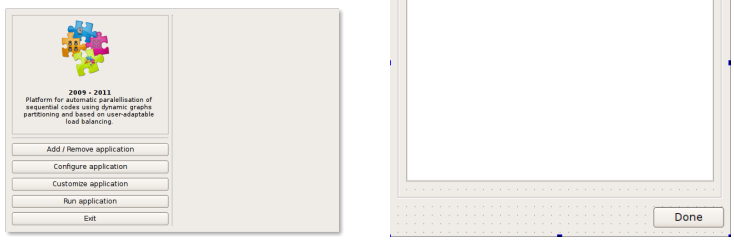
The parallelisation of this code is a computational challenge because, how can we design a parallel algorithm that sets up the information of the parallel processes that define themselves?

9.2.5 Improve the user interaction

If the framework has been designed to make end-users easy to run serial codes in a parallel way without the need of having a deep knowledge of parallel computing or even about the platform as a whole, it does not make sense to force the users to write tedious and strongly typed configuration files, where a huge number of parameters can be specified.

Although the platform is completely functional, the idea is to design a GUI (Graphical User Interface) to facilitate the task of writing the configuration files and another one to being able to see in real time the status of the execution of the user parallel algorithm, results achieved and access to the execution logs.

This GUI is nowadays on development but it still far away from its final and functional version (see figures 9.3(a) and 9.3(b)).



(a) Main window of the application. From the main menu, the application can be integrated within the platform and configured.

(b) Interface used to integrate a new application into the platform.

Figure 9.3: Graphical User Interface for configuring the application and managing the schemas (see section 1.4.2).

More important than the application GUI is to let the user to use his own meta-information in the file for configuring the applications. Meta-information refers to the ability of the framework to understand user structures with information about the problem to be run on the cluster but that has no be defined in the schema of the configuration file (see appendix B), such as for example, a way to define the useful range values to distinguish the figure from the ground (see explanation in section 9.2.1), or even special definitions for the algorithm.

9.3 New testbenchs

The proposed algorithms may offer little interest from the scientific point of view but they have been a very powerful tool as proof of concepts. Similarly, the main goal of the platform is not the algorithms that have been used as tests, but the parallelisation of these or even the techniques used depending on the scenario, but not the physical results of each algorithm as itself.

There are many areas where the parallel computing in very useful. Many of these fields are those related to mathematics, physics and engineering, but these are not all of them. For example, the group of Computer-assisted Paleoanthropology at the University of Zürich [107], earlier this year was asking for experts in parallel computing field. Other examples are medicine, chemistry, biology ... For this reason, there are many complex algorithms that can benefit from the platform and vice versa, the platform could benefit from using these algorithms, in order to further improve the implementation.

IV

Conclusiones y Trabajo futuro

10 Conclusiones

La plataforma balancea adecuadamente la carga de computación según aquellos *plugins* especificados por el usuario y descompone el problema según el método configurado de forma correcta, pudiéndose observar esto mediante la ejecución de las aplicaciones de prueba presentadas. Se observa que, independientemente del problema simulado usando la plataforma propuesta, los resultados a nivel de simulación son correctos, no perdiéndose información entre pasos del algoritmo ni detectándose valores erroneos de los resultados intermedios obtenidos. Las pruebas también han determinado que la interacción con usuario, a pesar de ser costosa, es correcta y permite la definición de datos, algoritmos y configuraciones necesarios para el correcto funcionamiento del algoritmo secuencial en la plataforma.

Desde el punto de vista del rendimiento de la plataforma, cabe destacar que el rendimiento de los algoritmos no se ha visto empeorado por el uso de la plataforma, salvo, como era de esperar, en el caso de la primera aplicación de pruebas explicada, la aplicación *Left*. Esto no es un fallo sino que, tal y como se explicó en la sección de resultados de dicha aplicación (ver capítulo 11).

La plataforma, tal y como se ha ido viendo a lo largo de los capítulos aquí expuestos, aún tratándose de una versión con algunos errores que deberán ser corregidos en futuras versiones, es perfectamente usable y cumple con los objetivos propuestos inicialmente.

De todas formas, no todas las conclusiones van a ser elogios hacia la plataforma. Esta aún tiene algunos fallos y puntos que deberían ser mejorados, con el fin de mejorar la interacción con el usuario. Como ejemplo, la aplicación del Sudoku sólo ha podido ser probada con Sudokus normales de 9×9 celdas, dado que el fichero de configuración para definir la rejilla ya ocupa más de 2000 lineas que tienen que ser escritas por el usuario, sin poder ser generadas de forma automatizada por algún programa externo o componente interno.

Tal y como se ha demostrado en el capítulo de resultados, la plataforma sigue tardando demasiado tiempo en tareas sencillas realizadas por el proceso maestro, como por ejemplo, la creación del grafo de aplicación o las tareas que involucran transacciones de red.

Como se puede observar por los resultados obtenidos, la red es el gran cuello de botella de la aplicación, haciendo inevitable el hecho de tener que optimizar todas las acciones que tienen que ver con la transferencia de información entre procesos. Este tema mejoraría ampliamente el rendimiento de las aplicaciones lo que a su vez daría mayor importancia al desarrollo de esta plataforma.

11

Trabajo futuro

Se dice que en un trabajo como el que aquí se presenta, el desarrollo realmente nunca finaliza, sólo se detiene. Esto es especialmente cierto en este caso, aún hay muchas cosas que probar, pulir, añadir ... pero en algún momento debe trazarse la línea final. En el caso de este trabajo, esta linea está en aquel punto en el que la plataforma era ya suficientemente funcional, cuando el trabajo no resultaba un esfuerzo extra para un programador familiarizado con los conceptos aquí expuestos.

Dado que el autor es muy consciente de los problemas y dificultades que plantean los diseños estáticos en cuanto a añadir nuevas funcionalidades, el diseño de la aplicación se ha realizado lo más modular posible, aspecto que puede observarse mediante el uso de plugins, clases abstractas..., lo que permite una fácil sustitución de casi cualquier parte del programa y facilitar la implementación e integración de cualquier comportamiento diferente o las mejoras de la plataforma.

Como punto final a este trabajo se exponen aquellos puntos que, no habiéndose terminado para el estado actual de esta documentación bien por considerarse puntos secundarios o bien por considerarse más alejados de la línea principal, comparten una relación directa con lo investigado hasta este momento. Las lineas de investigación y las mejoras propuestas requieren también de un largo estudio y esfuerzo en tiempo para poder llevarse a cabo y realizar las consecuentes pruebas. A pesar de que quizás solo algunas de estas mejoras se lleven a cabo, merece la pena listar aquí cada uno de los desarrollos alternativos. Dichos puntos pueden categorizarse en el siguiente esquema:

- Reducir las limitaciones de la plataforma.
- Añadir nuevas funcionalidades.
- Nuevos bancos de pruebas.

A continuación se detallan cada uno de estos puntos.

11.1 Reducir las limitaciones de la plataforma

Tal y como se explicó en el punto 1.4.1 la plataforma aún presenta diversas limitaciones que pueden resultar pequeños puzzles a los usuarios de esta plataforma.

Principalmente está el hecho de los fallos de memoria, en cuanto a que no toda la memoria que se reserva se libera, impidiendo así que se pueda ejecutar un mayor número de pasos de un mismo algoritmo aunque, obvio es, esto depende de la cantidad de memoria que el algoritmo en si necesite. Por otro lado y haciendo referencia a lo mismo, en contadas ocasiones sucede una violación de segmento (*core dump*, *segmentation fault*) si se usan los clásicos punteros en vez de usar los contenedores propios del lenguaje C++. Estos dos fallos están ampliamente relacionados y no se descarta que sea un mismo fallo presentando dos síntomas distintos. La solución a esto es simple aunque implica un mínimo cambio en la mentalidad del usuario de la plataforma a la hora de programar los algoritmos.

Desde un punto completamente diferente al anterior, existe una limitación seria que puede afectar al rendimiento de la plataforma en su globalidad; la creación y gestión dinámica de procesos. La integración de este método en la plataforma se basa en el uso de la primitiva implementada en la librería MPI-2, `MPI_Comm_spawn()` cuyo principal fin es la creación de procesos en caliente tras el lanzamiento de la aplicación MPI.

MPI cuenta en la actualidad con diversas versiones: la versión 1.3 (llamada MPI-1), cuya funcionalidad básica es el paso de mensajes y que cuenta con un entorno de ejecución estático, y la versión 2.2 (comúnmente llamada MPI-2), que incluye además nuevas funcionalidades como por ejemplo entrada salida paralela, gestión de procesos dinámicos y operaciones remotas.

Las especificaciones de MPI-2 describen tres interfaces para la creación dinámica de comunicadores, `MPI_Comm_spawn()`, `MPI_Comm_accept()` (mediante el uso de la primitiva `MPI_Comm_connect()`) y `MPI_Comm_join()`. La interfaz `MPI_Comm_spawn()` permite a un proceso MPI crear un número de instancias del citado proceso MPI. Los procesos recién lanzados forman un nuevo `MPI_COMM_WORLD` (MPI multicast) intracomunicador y puede comunicar con su proceso padre mediante el intercomunicador devuelto por la misma función.

Esta última mejora es quizás la más compleja e importante de todas pero, a pesar de que suena muy crítico, no todos los problemas mostrarán el mismo comportamiento una vez que esta solución haya sido implementada, unas mejorarán su rendimiento y otras por contra lo empeorarán. Además, dado que el software será más complejo que el actual, desde el punto de vista computacional, cabe esperar que inicialmente la plataforma falle más frecuentemente.

11.2 Añadir nuevas funcionalidades

Las plataforma aquí presentada cumple una serie de funcionalidades básicas desde el punto de vista de la ejecución paralela de código secuencial aunque, con el fin de continuar mejorando esta herramienta, se pretende mejorar los siguientes puntos:

11.2.1 Algoritmo de particionado

En cuanto a los algoritmos de particionado, se proponen las siguientes mejoras:

Descomposición funcional

Este trabajo se ha centrado, desde el punto de vista del particionado de la carga computacional, en técnicas de descomposición de dominio, dejando las técnicas de descomposición funcional como trabajo futuro.

Conviene recordar que las técnicas de descomposición de dominio son aquellas que parten un problema en grupos (dominios) a los que se aplica un mismo conjunto de instrucciones (algoritmo), mientras que las técnicas de descomposición funcional rompen un programa en tareas las cuales son aplicadas sobre un mismo conjunto de datos.

La aplicación de técnicas de descomposición funcional se presta muy bien para aquellos problemas que puedan ser fácilmente separados en diversas tareas.

Distinción entre “forma” y “fondo”

Los métodos estándares de particionamiento incluidos con el *framework* descomponen el problema en múltiples dominios sin tener en cuenta si la información contenida en el problema es “parte de la figura” o “parte del fondo” (ver imagen 9.1), conceptos también conocidos bajo los términos “espacio positivo” y “espacio negativo”.

¿Por qué debe particionarse también el “fondo” si ahí no hay información útil para el problema? y más aún, ¿por qué debe descomponerse la “figura” en dominios de igual forma si puede que en diversos puntos haya más información que en otros? La plataforma presentada no puede tener en cuenta esta situación dado que no hay manera aún de definir la meta-información necesaria para que el usuario defina “qué información es la útil para el algoritmo”. La única posibilidad reside en realizar este particionamiento de forma manual en los plugins de particionado pero esta opción no elimina del grafo aquellos vértices que no contengan información y éstos seguirían enviándose por red a otros procesos; aunque,

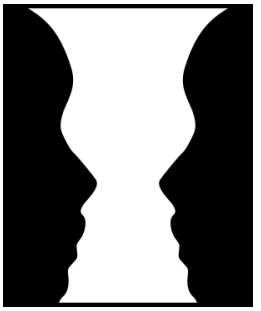


Figure 11.1: ¿Qué representa esta imagen? ¿Dos caras negras sobre un fondo blanco o una copa blanca sobre un fondo negro?

al estar la plataforma preparada para poder trabajar con bosques (se recomienda leer las definiciones de: forests y tree), la información nueva que pueda generarse tras cada paso del algoritmo puede ser tenida en cuenta para los siguientes pasos del algoritmo, facilitándose así la implantación de esta mejora al particionado y por tanto del rendimiento global del sistema.

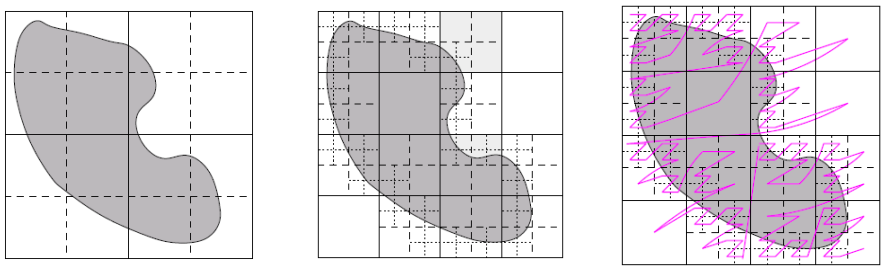
En relación a la segunda pregunta planteada recientemente, existen métodos recursivos [97] que consideran esta opción, como por ejemplo quadtree para problemas en 2D [98] (ver figura 11.2) u octree para aquellos definidos sobre espacios 3D. El comportamiento es similar a la segmentación en imágenes (leer la definición en segmentation) o a la detección de comunidades en grafos [99, 100].

Las nuevas tendencias mezclan las técnicas recursivas de particionado y las curvas "space filling curves" (ver imagen 11.2(c) y referencias en [101–103]).

Optimización de código

El uso de la librería *Metis*, ya citada anteriormente [55], para la descomposición del grafo de aplicaciones en múltiples dominios podría acelerar las acciones del proceso maestro, reduciendo probablemente el imbalanceo de carga de la aplicación y por tanto mejorando el rendimiento global de la plataforma.

Por otro lado, la librería de C++ BGL provee de una interfaz genérica para la creación, acceso y gestión de estructuras tipo grafo, ocultando los detalles de su implementación. Esta interfaz es una interfaz abierta en el sentido en que todas las librerías de grafos que implemente esta interfaz podrá interoperar con los algoritmos genéricos de BGL y con otros algoritmos que también usen esta interfaz. La interfaz BGL y sus componentes son completamente genéricos, al igual que los propocionados por STL [104]. El uso de esta librería podría optimizar y simplificar las tareas de tanto el proceso maestro como de los esclavos.



(a) Descomposición homogénea en quadtrees. Tanto la "figura" como el "fondo" están descompuestos con el mismo nivel de detalle

(b) Descomposición heterogénea de la superficie mediante quadtrees. La "figura" se descompone en cuadrantes con un nivel de detalles mayor que al del "fondo"

(c) Descomposición heterogénea de la superficie mediante quadtrees siguiendo una curva space filling curve. La "figura" se descompone en cuadrantes aplicando mayor detalle que al "fondo"

Figure 11.2: Dos métodos distintos de aplicación de quadtrees sobre una superficie genérica. El primero de ellos descompone la figura de forma igualitaria, mientras que el segundo de ellos da un mayor nivel de detalle sobre las zonas en las que ha sido definida la superficie

11.2.2 Tolerancia a fallos y recuperación tras errores

Los errores son inevitables, ningún desarrollo *software* está exento de errores. Incluso en el mejor de los escenarios, en condiciones ideales, la plataforma puede fallar, los procesos pueden agotar la memoria existente, la recepción de paquetes puede prolongarse hasta finalizar la plataforma por inanición (leer la definición en starvation). Esto es inherente a la programación y el uso de excepciones no es suficiente para prevenir la aparición de dichos errores más allá de las comprobaciones de condiciones de error y la evaluación de éstos antes de que sucedan.

Dado que no es posible la existencia de un programa informático sin errores, lo importante es el MTBF (término disponible en MTBF) y la minimización de las consecuencias de los errores, pero ¿cómo? Se debe garantizar que la ejecución de la plataforma continua correctamente tras uno o varios errores en uno o varios componentes del sistema. Esta explicación define el tan ampliamente comentado tema de la tolerancia a fallos (ver término en fault-tolerant). La tolerancia a fallos implica técnicas de aislamiento del componente fallido y la contención del fallo para prevenir su propagación a otros componentes del sistema. Desde el punto de vista del aislamiento del componente fallido, la plataforma toma la decisión de enviar datagramas sólo a aquellos procesos que hayan respondido correctamente y en tiempo a una petición anterior. Se considera que todos aquellos mensajes que no hayan sido recibidos por el proceso maestro de un proceso determinado, llegarán transcurrido un

tiempo infinito y por lo tanto, aplicando las fórmulas para calcular el número de tareas que se deben asignar a ese proceso en cuestión (ver fórmulas 4.1 y 4.2) se calcularán 0 tareas para dicho proceso. Aunque la plataforma cumple las primera de las condiciones de tolerancia a fallos, no se puede considerar que sea completamente tolerante, dado que el segundo requisito, "contención del fallo para prevenir su propagación a otros componentes del sistema", en la versión actual no se cumple.

El término "recuperación tras errores" está básicamente relacionado con la propiedad de tolerancia a fallos, dado que consiste en una serie de procedimientos para recuperar aquella información ausente o errónea tras la aparición del error. La recuperación tras errores intenta resolver las preguntas que se plantean en la siguiente situación: "supóngase un cluster formado por diversas máquinas en la cual se está corriendo un determinado algoritmo. El problema se parte en dominios que se envían a cada proceso esclavo. Los esclavos ejecutan el algoritmo sobre el dominio recibido para obtener los resultados del siguiente paso de la simulación. Cada vez que un proceso termina sus cálculos, envía los resultados al proceso maestro transfiriéndose estos a través de la interfaz de red configurada para tal efecto. Uno de los esclavos, por un fallo de red, no puede contactar con el proceso maestro y, por tanto, no puede enviar sus resultados. Mientras el proceso maestro recibe los resultados enviados por los procesos esclavos, excepto aquellos del proceso cuya interfaz de red falla. En este punto, el proceso maestro no puede esperar hasta recibir los datos del proceso ausente pero, dichos datos deberán ser ejecutados por el resto de procesos esclavos que sí tienen una conexión activa con el proceso maestro, de forma que se pueda continuar con la ejecución normal del algoritmo". Las políticas que determinan el comportamiento de la plataforma en situaciones similares a la anterior se corresponden con las técnicas de recuperación frente a errores. El estado actual de la plataforma no tiene en cuenta estas situaciones dejándose la recuperación tras errores como un punto para el trabajo futuro relacionado con la plataforma.

11.2.3 Mejorar el rendimiento de la operación de salvado

Tal y como ha sido mencionado en diversas ocasiones en este documento, las operaciones de E/S (entrada/salida, I/O en lengua inglesa) y las operaciones de comunicación son las acciones que, siendo procedimientos no paralelos, empeoran el rendimiento de la aplicación paralela (en nuestro caso, el rendimiento de la plataforma). La minimización de las comunicaciones entre procesos es tarea del algoritmo de particionado y por tanto, responsabilidad del usuario final del sistema pero, en cuanto a la minimización de las operaciones de E/S, el usuario puede hacer bien poco.

La E/S es prácticamente el mayor cuello de botella en muchas aplicaciones paralelas. La razón principal del bajo rendimiento de las aplicaciones debido a la E/S se debe a que

la E/S paralela está optimizada para accesos a grandes volúmenes de datos (tasas en el orden de los MB) mientras que las aplicaciones paralelas generalmente sólo hacen uso de volúmenes de datos de E/S de entorno a los kB o menos.

Una forma de mejorar el rendimiento de las aplicaciones paralelas es mejorando el rendimiento del proceso de salvado de resultados, procedimiento ejecutado por el proceso maestro según el algoritmo de salvado especificado por el usuario (ver página 13). Las operaciones de E/S pueden realizarse en paralelo aprovechando las funcionalidades de las primitivas de MPI–I/O.

La funcionalidad principal de las primitivas de E/S es que permite a los usuarios acceder a bloques de datos no contiguos en ficheros mediante una única llamada a esta extensión de MPI definiendo las entradas del fichero como tipos derivados. Las funcionalidades de la entrada/salida paralelas se refieren a aquel conjunto de funciones diseñadas para abstraer el manejo de las operaciones de E/S en sistemas distribuidos y permitir y facilitar el acceso a fichero usando los tipos derivados ya existentes.

Los tipos derivados en MPI pueden ser de dos tipos distintos: tipos básicos, aquellos que se corresponden con los tipos básicos de casi cualquier lenguaje de programación y tipos derivados, creados mediante constructores y que constan de un conjunto de datos básicos que pueden estar situados contigua o no contíguamente.

Esta mejora aún no ha sido implementada dado la poca investigación que se ha hecho sobre este tema y la dificultad para obtener un buen rendimiento usando estas primitivas [105]. Por ejemplo, existen diversas implementaciones sobre multiplicaciones de matrices dispersas usando las librerías de MPI–I/O que demuestran rendimiento completamente desastrosos [106].

11.2.4 Optimizar los procedimientos del proceso maestro

En relación al tema expuesto en el apartado anterior, existe también un problema de rendimiento en los procesos que se ejecutan del lado del maestro. La ejecución del maestro es secuencial y aún no ha sido paralelizado.

El procedimiento del proceso maestro que consume mayor tiempo es el código para la creación de los distintos grafos (grafo de aplicación y grafo del sistema) a partir de los valores y configuraciones especificados por el usuario en los correspondientes ficheros de configuración. La complejidad de este algoritmo crece como $O(N \times M)$ donde N identifica el número de vértices del grafo y M es la cardinalidad del mayor conjunto de enlaces de cada uno de los vértices del grafo.

Paralelizar este código supone un verdadero reto dado que, ¿cómo diseñamos un algoritmo paralelo que define la información de los propios procesos paralelos de la plataforma?

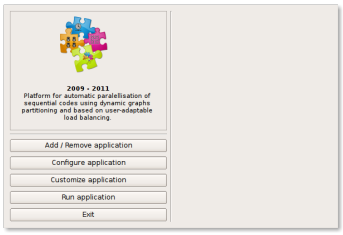
11.2.5 Mejorar el manejo por parte del usuario

Si la plataforma fue diseñada para facilitarle a los usuarios finales el poder ejecutar código secuencial en una manera paralela sin que éstos tengan que tener conocimientos sobre computación paralela o incluso sobre la propia plataforma, no tiene sentido que se les obligue a escribir ficheros de configuración tan fuertemente tipados donde se pueden especificar gran cantidad de parámetros.

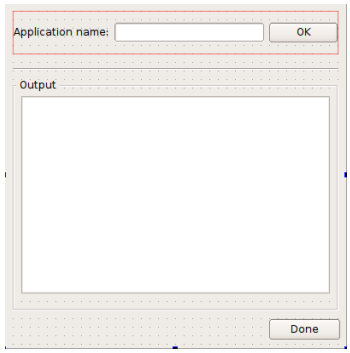
Aunque la plataforma en su estado actual es completamente funcional, se propone también diseñar una interfaz gráfica de usuario GUI que facilite al usuario la tarea de escribir los ficheros de configuración y que por otra parte permite también ver en tiempo real el estado de la plataforma, del algoritmo en ejecución, los resultados ya obtenidos o incluso acceder a los logs de ejecución.

Esta interfaz de usuario está actualmente en desarrollo aunque aún está lejos de ser una versión estable y funcional (ver imágenes 11.3(a) y 11.3(b)).

Más importante que la interfaz de usuario sería dotar al usuario de una funcionalidad para que pueda especificar su propia meta-información en el fichero de configuración del algoritmo. El concepto de meta-información hace referencia a la capacidad de la plataforma de entender las estructuras de usuario con la posible información acerca del algoritmo a ejecutar en el cluster y que no tenga ésta que estar definida en los esquemas de los ficheros de configuración (ver apéndice B), como por ejemplo, una forma de poder definir los valores que distinguen la forma del fondo (ver explicación en la sección 11.2.1), o incluso definiciones especiales para el algoritmo en cuestión.



(a) Aspecto de la ventana principal de la aplicación. Desde el menu principal se puede integrar la aplicación en la plataforma y configurarla



(b) Interfaz de integración de una nueva aplicación en la plataforma.

Figure 11.3: Interfaz gráfica para la configuración de la aplicación y el manejo de los *schemas* (ver sección 1.4.2)

11.3 Nuevos bancos de pruebas

Los algoritmos propuestos quizás ofrecen escaso interés desde el punto de vista científico pero han sido una herramienta muy poderosa como prueba de conceptos. Igualmente, el principal objetivo de la plataforma no es los algoritmos que se han usado como prueba, si no la paralelización de estos, las técnicas usadas según el escenario aunque no los resultados físicos de los algoritmo en si.

Existen multitud de campos en los que la computación paralela en de gran utilidad. Muchos de estos campos son aquellos relacionados con las matemáticas, física e ingenierías, pero esos no son todos. Así por ejemplo, el grupo de Paleoantropología asistida por ordenador (Computer-assisted Paleoanthropology) de la Universidad de Zúrich [107] pedía a principios de año expertos en computación paralela para el desarrollo de un proyecto. Otros ejemplos son medicina, química, biología... Por este motivo, existen multitud de algoritmos complejos que pueden beneficiarse de la plataforma y viceversa, la plataforma podría beneficiarse del uso de dichos algoritmos, con el fin de poder seguir mejorando la aplicación.

V

Bibliography

[1] A. Lastovetsky, "Scientific programming for heterogeneus systems - bridging the gap between algorithms and applications," in *PARELEC'06 IEEE Proceedings*, pp. 3–8, 2006. (document)

[2] R. Reddy, *HMPI: A Message-Passing Library for Heterogeneous Networks of Computers*. PhD thesis, Computer Science Department, University College Dublin, Belfield, Dublin., 2005. 1.2

[3] V. S. Sunderam, "PVM: a framework for parallel distributed computing," *Concurrency, Practice and Experience*, vol. 2, no. 4, 1990. 1.2

[4] M. P. I. Forum, "MPI: A message-passing interface standard," tech. rep., 1994. 1.2, 2.1

[5] G. E. Fagg and J. Dongarra, "PVMPI: An integration of PVM and MPI systems," *Calculateurs Parallèles*, vol. 8, no. 2, pp. 151–166, 1996. 1.2

[6] A. Lastovetsky and R. Reddy, "HeteroMPI: Towards a message-passing library for heterogeneous networks of computers," *Journal of Parallel and Distributed Computing*, 2005. 1.2

[7] L. Huang, B. Chapman, and R. Kendall, "Openmp for clusters," in *The Fifth European Workshop on OpenMP*, pp. 22–26, 2003. 1.2

[8] A. Lastovetsky and R. Reddy, "Data partitioning with a realistic performance model of networks of heterogeneous computers," *Scientific Programming*, vol. 2, no. 13, pp. 93–112, 2005. 1.3, 2.2

[9] R. Grimaldi, *Discrete and combinatorial mathematics*. Addison-Wesley Reading, Mass, 1994. 1.3

[10] K. Rosen and J.G.Michaels, *Handbook of discrete and combinatorial mathematics*. CRC Press Boca Raton, Fla, 2000. 1.3

[11] W. Pree, "Meta patterns: A means for capturing the essentials of reusable object-oriented design," *Proceedings of the 8th European Conference on Object-Oriented Programming*, pp. 150–162, 1994. 1.4

[12] R. Agrawal and J. C. Shafer, "Parallel mining of association rules," *IEEE Trans. on Knowl. and Data Eng.*, vol. 8, pp. 962–969, 1996. 1.4

[13] H. Eui-Hong, G. Karypis, and V. Kumar, "Scalable parallel data mining for association rules," *Knowledge and Data Engineering, IEEE Transactions on*, vol. 12, no. 3, pp. 337–352, 2000. 1.4

[14] G. S. Almasi and A. Gottlieb, *Highly parallel computing*. Benjamin-Cummings Publishing Co., Inc., 1989. 2.1

[15] D. Kirk and W. mei Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann, 2010. 2.1

[16] D. A. Patterson and J. L. Hennessy, *Computer organization and design: the hardware/-software interface*. Morgan Kaufmann Publishers Inc., 1998. 2.1

[17] I. Foster, *Designing and Building Parallel Programs*. Addison-Wesley Publishing, 1995. 2.1, 2.2

[18] K. Chandy and J. Misra, *Parallel Programming Design*. 2.1

[19] J. Jaja, *An introduction to parallel algorithms*. Addison Wesley Longman Publishing Co., Inc., 1992. 2.1

[20] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the April 18-20, 1967, spring joint computer conference*, pp. 483–485, 1967. 2.1

[21] A. Tanenbaum, *Redes de computadores*. Prentice-Hall, 1997. 2.1

[22] J. Dongarra and A. Lastovetsky, *An overview of heterogeneous high performance and grid computing*. American Scientific Publishers, 2006. 2.1

[23] A. Gidenstam, B. Koldehofe, M. Papatriantafilou, and P. Tsigas, "Dynamic and fault-tolerant cluster management," tech. rep., 2005. 2.1

[24] M. J. Flynn, "Very high-speed computing systems," *Proceedings of the IEEE*, vol. 54, no. 12, pp. 1901–1909, 1966. 2.1

[25] E. Dijkstra, *The origin of concurrent programming: From semaphores to remote procedure calls*, pp. 198–227. Springer-Verlag New York, Inc., 2002. 2.1

[26] R. Perrott, *Parallel Programming*. Addison-Wesley Publishing Company, 1987. 2.1

[27] "Posix: Ieee 1003.1." 2.1

[28] *OpenMP shared memory parallel programming*, vol. 2104, 2001. 2.1

[29] C. High Performance Fortran Forum, "High performance fortran language specification," *SIGPLAN Fortran Forum*, vol. 13, pp. 22–55, 1994. 2.1

[30] H. P. F. Forum, "High performance fortran language specification," 1994. 2.1

[31] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications ACM*, vol. 51, pp. 107–113, 2008. 2.1

[32] T. Farias, J. M. Teixeira, P. Leite, G. Almeida, M. W. S. Almeida, V. Teichrieb, and J. Kelner, "High performance computing: Cuda as a supporting technology for next generation augmented reality applications," *Revista de Informática Teórica e Aplicada*, vol. 16, no. 1, pp. 63–88, 2009. 2.1

[33] J. Ekanayake and G. Fox, "High performance parallel computing with clouds and cloud technologies," vol. 34, pp. 20–38, Springer Berlin Heidelberg, 2010. 2.1

[34] "Hadoop." 2.1

[35] K. Dowd, "High performance computing with cuda," *Performance Computing*, vol. 1, no. 2, pp. 275–295, 2009. 2.1

[36] M. O. Ball, "Computing network reliability," 1979. 2.2

[37] J. S. Provan and M. O. Ball, "The complexity of counting cuts and of computing the probability that a graph is connected," *SIAM Journal on Computing*, no. 12, pp. 777–788, 1983. 2.2

[38] D. W. Corne, M. J. Oates, and G. D. Smith, *Telecommunications Optimization*. Wiley, 2000. 2.2

[39] W. Pedrycz and A. Vasilakos, *Computational intelligence in telecommunications net-*

works. CRC Press, 2001. 2.2

[40] R. Diekmann, B. Monien, and R. Preis, "Load balancing strategies for distributed memory machines," in *Multi-Scale Phenomena and Their Simulation*, pp. 255–266, World Scientific, 1997. 2.2

[41] R. Canal, J. M. Parcerisa, and A. González, "Dynamic cluster assignment mechanisms," in *HPCA-6*, pp. 132–142, 2000. 2.2

[42] R. Bhargava and L. K. John, "Improving dynamic cluster assignment for clustered trace cache processors," tech. rep., 2003. 2.2

[43] K. Amiri, D. Petrou, G. Ganager, and G. Gibson, "Dynamic function placement in active storage clusters," tech. rep., 1999. 2.2

[44] A. Lastovetsky and J. Twamley, "Towards a realistic performance model for networks of heterogeneous computers," tech. rep., 2005. 2.2

[45] D. A. Bacigalupo, S. A. Jarvis, L. He, D. P. Spooner, and G. R. Nudd, "Comparing layered queuing and historical performance models of a distributed enterprise application.," in *IASTED International Conference on Parallel and Distributed Computing and Networks*, pp. 608–613, 2005. 2.2

[46] D. A. Bacigalupo, S. A. Jarvis, L. He, D. Spooner, D. Pelych, and G. R. Nudd, "A comparative evaluation of two techniques for predicting the performance of dynamic enterprise systems.," in *PARCO*, pp. 163–170, 2005. 2.2

[47] S. Kumar, S. K. Das, and R. Biswas, "Graph partitioning for parallel applications in heterogeneous grid environments," in *IPDPS '02: Proceedings of the 16th International Symposium on Parallel and Distributed Processing*, p. 66, 2002. 2.3

[48] B. Hendrickson and R. Leland, "An improved spectral graph partitioning algorithm for mapping parallel computations," *SIAM J. Sci. Comput.*, vol. 16, pp. 452–469, 1995. 2.3

[49] A. Pothen, H. D. Simon, and K.-P. Liou, "Partitioning sparse matrices with eigenvectors of graphs," *SIAM J. Matrix Anal. Appl.*, vol. 11, pp. 430–452, 1990. 2.3

[50] G. L. M. S. Teng, W. Thurston, and S. A. Vavasis, "Automatic mesh partitioning," tech. rep., 1992. 2.3

[51] G. L. Miller, S.-H. Teng, and S. A. Vavasis, "A unified geometric approach to graph separators," in *Proceedings of the 32nd annual symposium on Foundations of computer science*, pp. 538–547, 1991. 2.3

[52] B. Hendrickson and R. Leland, "A multilevel algorithm for partitioning graphs," in *Proceedings of the 1995 ACM/IEEE conference on Supercomputing*, 1995. 2.3

[53] N. Mansour, R. Ponnusamy, A. Choudhary, and G. Fox, "Graph contraction for physical optimization methods: a quality-cost tradeoff for mapping data on parallel computers," in *Proceedings of the 7th international conference on Supercomputing*, pp. 1–10, 1993. 2.3

[54] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM Journal on Scientific Computing*, vol. 20, pp. 359–392, 1998.

2.3

[55] "Metis website." 2.3, 9.2.1, 11.2.1

[56] "Parmetis website." 2.3

[57] C. Walshaw and M. Cross, "Mesh partitioning: a multilevel balancing and refinement algorithm," 1998. 2.3

[58] "The chaco's users guide," 1995. 2.3

[59] R. Preis and R. Diekmann, "Party - a software library for graph partitioning," in *Advances in Computational Mechanics with Parallel and Distributed Processing*, pp. 63–71, Civil-Comp Press, 1997. 2.3

[60] S. Schamberger and J. michael Wierum, "Graph partitioning in scientific simulations: Multilevel schemes versus space-filling curves," 2003. 2.3

[61] C.-W. Ou and S. Ranka, "Parallel incremental graph partitioning," 1997. 2.3

[62] E. Lusk, N. Doss, and A. Skjellum, "A high-performance, portable implementation of the mpi message passing interface standard," *Parallel Computing*, vol. 22, pp. 789–828, 1996. 3.1.4

[63] J. L. Träff, "Implementing the mpi process topology mechanism," in *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pp. 1–14, IEEE Computer Society Press, 2002. 3.2.1

[64] S. Nesmachnow, *Algoritmos genéticos paralelos y su aplicación al diseño de redes de comunicaciones confiables*. PhD thesis, Instituto de Computación, Facultad de Ingeniería, Universidad de la República Montevideo, Uruguay, 2004. 4.1

[65] G. Zumbusch, "On the quality of space-filling curve induced partitions," *Z. Angew. Math. Mech*, vol. 81, 2000. 4.2.1

[66] A. K. Patra and J. T. Oden, "Computational techniques for adaptive hp finite element methods," *Finite Elements in Analysis and Design*, vol. 25, 1997. 4.2.1

[67] A. K. Patra, A. Laszloffy, and J. Long, "Data structures and load balancing for parallel adaptive hp finite-element methods," *Computers and Mathematics with Applications*, vol. 46, 2003. 4.2.1

[68] F. Guenther, M. Mehl, M. Poegl, and C. Zenger, "A cache-aware algorithm for pdes on hierarchical data structures based on space-filling curves," *SIAM J. Sci. Comput.*, vol. 28, pp. 1634–1650, 2006. 4.2.1

[69] A. Heinecke and M. Bader, "Parallel matrix multiplication based on space-filling curves on shared memory multicore platforms," in *Proceedings of the 2008 workshop on Memory access on future processors: a solved problem?*, pp. 385–392, 2008. 4.2.1

[70] M. Bader, R. Franz, S. Guenther, and A. Heinecke, "Hardware-oriented implementation of cache oblivious matrix operations based on space-filling curves," in *Parallel Processing and Applied Mathematics*, vol. 4967, pp. 628–638, 2008. 4.2.1

[71] B. Moon, H. Jagadish, C. Faloutsos, and J. H. Saltz, "Analysis of the clustering properties of the hilbert space-filling curve," *IEEE Trans. on Knowl. and Data Eng.*, vol. 13,

pp. 124–141, 2001. 4.2.1

[72] C. H. H. A. Rau-Chaplin, "Compact hilbert indices: Space-filling curves for domains with unequal side lengths," *Inf. Process. Lett.*, vol. 105, pp. 155–163, 2008. 4.2.1

[73] M. F. Mokbel and W. G. Aref, "Irregularity in high-dimensional space-filling curves," *Distributed Parallel Databases*, vol. 29, pp. 217–238, 2011. 4.2.1

[74] D. J. Abel and D. M. Mark, "A comparative analysis of some two-dimensional orderings," *Int. J. Geographical Information Systems*, vol. 4, pp. 21–31, 1990. 4.2.1

[75] M. Geimer, F. Wolf, B. J. N. Wylie, E. Ábrahám, D. Becker, and B. Mohr, "The scalasca performance toolset architecture," *Concurrency and Computation: Practice and Experience*, vol. 22, pp. 702–719, 2010. 4.3, 7

[76] "Paraver: Obtain detailed information from raw performance traces." 4.3

[77] S. Benedict, V. Petkov, and M. Gerndt, "Periscope: An online-based distributed performance analysis tool," in *Tools for High Performance Computing 2009*, pp. 1–16, 2010. 4.3

[78] H. Brunst, M. Winkler, W. Nagel, and H.-C. Hoppe, "Performance optimization for large scale computing: The scalable vampir approach," in *Computational Science - ICCS 2001*, vol. 2074, pp. 751–760, 2001. 4.3

[79] M. Gardner, "The fantastic combinations of john conway's new solitaire game "life"," *Scientific American*, vol. 223, pp. 120–123, 1970. 6.2

[80] B. Chopard, P. Luthi, and A. Masselot, "Cellular automata and lattice boltzmann techniques: An approach to model and simulate complex systems," in *ADVANCES IN PHYSICS*, p. 98, 1998. 6.2.1

[81] R. B. Potts, "Some generalized order-disorder transformations," vol. 48, pp. 106–109, Proceedings of the Cambridge Philosophical Society, 1952. 6.3

[82] E. Ising, "Beitrag zur theorie des ferromagnetismus," *Journal of Physics*, vol. 31, 1925. 6.3

[83] A. Alexander, C. Mark, and R. K. A, *Single-Cell-Based Models in Biology and Medicine*. Birkhäuser Verlag AG., 2007. 6.3, 6.3.2

[84] F. Graner and J. A. Glazier, "Simulation of biological cell sorting using a 2-dimensional extended potts model," *Physics Review L*, vol. 69, no. 13, pp. 2013–2016, 1992. 6.3

[85] N. J. Poplawski, M. Swat, J. S. Gens, and J. A. Glazier, "Adhesion between cells, diffusion of growth factors, and elasticity of the aer produce the paddle shape of the chick limb," *Physica A: Statistical and Theoretical Physics*, vol. 373, pp. 521–532, 2007. 6.3

[86] S. Turner and J. A. Sherratt, "Intercellular adhesion and cancer invasion: A discrete simulation using the extended potts model," *Journal of Theoretical Biology*, vol. 216, no. 1, pp. 85–100, 2002. 6.3

[87] N. Metropolis, A. Rosenbluth, M. Rosenbluth, A. H. Teller, and E. Teller, "Equation of state calculations by fast computing machines," *J. Chem. Phys*, vol. 21, no. 13,

pp. 1087–1092, 1953. 6.3.2, 7.3

[88] N. Chen, J. A. Glazier, J. A. Izaguirre, and M. S. Albera, "A parallel implementation of the cellular potts model for simulation of cell-based morphogenesis.," *Comput Phys Commun*, vol. 176, no. 11–12, pp. 670–681, 2007. 6.3.2, 6.3.2

[89] F. P. Cercato, J. C. M. Mombach, and G. G. H. Cavalheiro, "High performance simulations of the cellular potts model," *High Performance Computing Systems and Applications*, vol. 0, p. 28, 2006. 6.3.2

[90] E. Gusatto, J. C. Mombach, F. P. Cercato, and G. H. Cavalheiro, "An efficient parallel algorithm to evolve simulations of the cellular potts model," *Parallel Processing Letters*, vol. 15, no. 1–2, pp. 199–208, 2005. 6.3.2

[91] E. E. Santos and G. Muthukrishnan, "Efficient simulation based on sweep selection for 2-d and 3-d ising spin models on hierarchical clusters," *ipdps*, vol. 14, p. 229b, 2004. 6.3.2

[92] J. Castellanos, C. Martín-vide, V. Mitrana, and J. M. Sempere, "Networks of evolutionary processors," in *Acta Informatica*, pp. 401–412, Springer-Verlag, 2003. 6.4

[93] E. del Rosal García, J. M. R. Siles, R. N. Hervás, C. C. Marroquín, and A. O. de la Puente, "On the solutions of np-complete problems by means of jnep run on computers," in *ICAART*, pp. 605–612, 2009. 6.4

[94] Y. Chen, H. Ohashi, and M. Akiyama, "Thermal lattice bhatnagar-gross-krook model without nonlinear deviations in macrodynamic equations," *Physics Review E*, vol. 50, no. 4, pp. 2776–2783, 1994. 6.6.1

[95] Y. H. Qian, S. Succi, and S. A. Orszag, "Recent advances in lattice boltzmann computing," 1995. 6.6.2

[96] "Hochleistungsrechenszentrum bayern." 7.2.1, 7.4.1

[97] Q. F. Stout, D. L. D. Zeeuw, T. I. Gombosi, C. P. Groth, H. G. Marshall, and K. Kenneth G. Powell, "Adaptive blocks: a high performance data structure," in *Proceedings of the 1997 ACM/IEEE conference on Supercomputing*, pp. 1–10, 1997. 9.2.1, 11.2.1

[98] S. Wang and M. P. Armstrong, "A quadtree approach to domain decomposition for spatial interpolation in grid computing environments," *Parallel Computing*, vol. 29, pp. 1481–1504, 2003. 9.2.1, 11.2.1

[99] S. Fortunato, "Community detection in graphs," *Physics Reports*, vol. 486, pp. 75–174, 2010. 9.2.1, 11.2.1

[100] G. E. Martin, *Polyominoes: a guide to puzzles and problems in tiling*. Mathematical Association of America, 1991. 9.2.1, 11.2.1

[101] P. M. Campbell, K. D. Devine, J. E. Flaherty, L. G. Gervasio, and J. D. Teresco, "Dynamic octree load balancing using space-filling curves," tech. rep., Williams College Department of Computer Science, 2003. 9.2.1, 11.2.1

[102] P. Ajmera, R. Goradia, S. Chandran, and S. Aluru, "Fast, parallel, gpu-based space filling curves and octrees." 9.2.1, 11.2.1

[103] J. M. Montesinos, *Classical tessellations and three-manifolds*. Springer, 1987. 9.2.1, 11.2.1

[104] M. H. Austern, *Generic programming and the STL: using and extending the C++ Standard Template Library*. Addison-Wesley, 1998. 9.2.1, 11.2.1

[105] R. Thakur, W. Gropp, and E. Lusk, "A case for using mpi's derived datatypes to improve i/o performance," in *Proceedings of the 1998 ACM/IEEE conference on Supercomputing*, pp. 1–10, IEEE Computer Society, 1998. 9.2.3, 11.2.3

[106] M. van Hulte, "Parallelization of a sparse matrix-vector multiplication algorithm," tech. rep., 2006. 9.2.3, 11.2.3

[107] "http://www.aim.uzh.ch/morpho/wiki/." 9.3, 11.3

Lists

List of algorithms

List of codes

List of equations

List of figures

List of tables

List of text boxes

ACRONYMS

ANSI . . . . . American National Standards Institute

API . . . . . Application Program Interface

BFS . . . . . Breadth First Search

BGK . . . . . Bhatnagher-Gross-Krook

BGL . . . . . Boost Graph Library

BvK . . . . . Born-von Karman

CA . . . . . Cellular Automata

CFD . . . . . Computational Fluid Dynamics

CPM . . . . . Cellular Potts Model

CPU . . . . . Central Processing Unit

CUDA . . . . . Compute Unified Device Architecture

DDM . . . . . Domain Decomposition Methods

DFS . . . . . Depth First Search

FCFS . . . . . First-Come, First-Servered

GoL . . . . . Game of Life

GPU . . . . . Graphics Processing Unit

GUI . . . . . Graphical User Interface

HLRB . . . . . Hochleichtungrechnenzentrum Bayern

HNoC . . . . . Heterogeneous Network of Computers

HPC . . . . . High Performance Computing

HPF . . . . . High Performance Fortran

HPP . . . . . Hamiltonian Path problem

ICMP . . . . . Internet Control Message Protocol

JNEP . . . . . Java Networks of Evolutionary Processors

LBM ..... Lattice Boltzmann Methods

LCFS ..... Last-Come, First-Servered

LGCA ..... Lattice-Gas Cellular Automata

LIS ..... Language Independent Specifications

MCS ..... Montecarlo Step

MIMD ..... Multiple Instruction, Multiple Data

MISD ..... Multiple Instruction, Single Data

MPI ..... Message Passing Interface

MPMD ..... Multiple Program Multiple Data

MTBF ..... Mean Time Between Failures

NEP ..... Networks of Evolutionary Processors

NoC ..... Network of Computers

NP ..... Non-deterministic Polynomial time

NUMA ..... Non-Uniform Memory Access

OSI ..... Open System Interconnection

POSIX ..... Portable Operating System Interface

PVM ..... Parallel Virtual Machine

RMS ..... Resources Management System

RO ..... Read only

RW ..... Read write

SIMD ..... Single Instruction, Multiple Data

SIRO ..... Service In Random Order

SISD ..... Single Instruction, Single Data

SPACE FILLING ....
Curves whose range contains the entire 2D unit square or more generally, a N–dimensional hypercube. Because Giuseppe Peano (1858–1932) was the first to discover one, space-filling curves in the 2-dimensional plane are commonly called Peano curves

SPMD ..... Single Program Multiple Data

STL ..... Standard Template Library

TCP ..... Transmission Control Protocol

UDP ..... User Datagram Protocol

UMA ..... Uniform Memory Access

VLSI ..... Very Large Scale Integration

XML ..... Extensible Markup Language

TERMINOLOGY

abstract

Class whose behaviour is defined as "generic" and much of it definitions will be undefined and unimplemented

backtracking

Algorithm to incrementally build candidates to the solutions set and reject them as soon as it determines that it cannot be taken as a valid solution

bandwidth

Amount of data that can be communicated per unit of time. Commonly expressed as megabytes/sec or gigabytes/sec

big-O

Notation used to describe the limiting behaviour of a function when the argument tends towards a particular value or infinity. In computer science, it is used to classify algorithms by their computational complexity

cardinality

A measure of the "number of elements of the set"

class

Construct that consists of and is composed from structural and also behavioural constituents and can be instanced

cluster

Group of linked computers, working together closely thus in many respects forming a single computer

communication

The way parallel tasks exchange data. There are several ways this can be accomplished, such as through a shared memory bus or over a network

communicator

Object that connects groups of processes in a MPI session. Each communicator gives each contained process an independent identifier and arranges its contained processes in an ordered topology

constructor

Special object oriented method called at the creation of an object. It prepares the new object for use, often accepting parameters which the constructor uses to set any member variables required when the object is first created

deserialisation

Opposite operation of serialisation, extracting a data structure from a series of bytes. It is also known as inflating or unmarshalling

destructor

Method which is automatically invoked when the object is destroyed. Its main purpose is to clean up and to free the resources which were allocated by the object along its life cycle and unlink it from other objects or resources invalidating any references in the process

distributed memory

Refers to network based memory access for physical memory. As a programming model, tasks can only logically "see" local machine memory and must use communications to access memory on other machines where other tasks are executing

dynamic library

Libraries which its subroutines are load into an application at load-time or run-time, rather than linking them at compile-time

edge

Link that connect pairs of vertices

exception

Programming interrupt. Special conditions that change the normal flow of program execution

fault-tolerant

Property that enables systems to continue operating properly in the event of the failure of one or more faults within some of its components

forests

A forest (or forest of graphs) is a disjoint union of trees

framework

Abstraction in which software providing generic functionality can be selectively changed by user code, thus providing application specific software. It is a collection of software libraries providing a defined application programming interface

granularity

Parameter that measures the amount of work done by each processor

graph

Abstract representation of a set of objects where some pairs of them are connected with others. The interconnected objects are represented by mathematical abstractions called vertices, and the links that connect pairs of vertices are called edges

grid

Group of linked computers that tend to be more loosely coupled, heterogeneous, and geographically dispersed than clusters

heterogeneous

Cluster that is not homogeneous

homogeneous

Cluster where all its nodes have the same architecture, operating system, hardware features and key component libraries

load imbalance

Uneven distribution of work across cores

intercommunicator
Communicator used to transfer packages between MPI processes of different groups

interface
Pure virtual class that provides required information to modules from a transparent point of view

intracommunicator
Communicator used to transfer packages between MPI processes of the same group

latency
Time taken to send a minimal (0 byte) message from point A to point B. It is commonly expressed as microseconds

library
Collection of resources used to develop software. It contains code and data that provide services to independent programs

Lindenmayer
Rewriting system, variant of a formal grammar, where all the non-terminal characters of a correct word must be replaced at the same iteration

macro
Rule that specifies how a certain input sequence should be mapped to an output sequence according to a defined procedure. The difference between macros and procedures resides on the use of the stack segment: macros do not use stack while the definition of a procedure or function implies the existence of a stack

makefile
File which specifies hoy to derive target programs used by the utility that automatically builds executable programs and libraries from source code

marshalling
See serialisation

master
Process that performs synchronization tasks and gathering the results obtained by the slave processes. It runs on a node called "master node"

message passing
Paradigm of communication where messages are sent from a sender to one or more recipients. Forms of messages include (remote) method invocation, signals, and data packets

method
Functions in object oriented languages

middleware
Software that provides a link between separate software applications. It connects two or more applications and passes data between them

Montecarlo method

Class of computational algorithms that rely on repeated random sampling to compute their results

MTBF

Predicted elapsed time between inherent failures of a system during operation. It can be calculated as the average time between failures of a system

namespaces

Abstract containers created with the aim of holding a logical grouping of unique identifiers or symbols

neighbourhood

Spatial region around each cell used to compute the next state

node

A standalone computer. They are networked together to comprise a supercomputer

octree

Tree data structure in which each internal node has exactly eight children. They are most often used to partition a three dimensional space by recursively subdividing it into eight octants

parallel overhead

The amount of time required by a parallel program to coordinate parallel tasks, as opposed to the time spent on doing useful work. It includes, among others, synchronisation procedures, data communications and task termination times

path

Sequence of vertices of a graph such that from each of the vertices there is an edge to the next vertex in the sequence

plugin

Set of software components that adds specific abilities to a larger software application. If supported, plugins enable customizing the functionality of an application. They are also knows under the name “plug-in”

portability

Codebase feature to be able to reuse the existing code instead of creating new code when moving software from an environment to another

preprocessing macro

macro that is replaced by a preprocessor or a compiler with a char-string value before compilation

pseudocode

Compact and informal high-level description of a computer programming algorithm that uses the structural conventions of a programming language, but is intended for human reading rather than machine reading. It typically omits details that are not essential for human understanding of the algorithm, such as variable declarations and system-specific code

quadtree

Tree data structure in which each internal node has exactly four children. They are

most often used to partition a two dimensional space by recursively subdividing it into four quadrants or regions

reliability

Measure to assess the probability of a successful communication between pairs of nodes

scalability

Ability of a program to demonstrate a proportionate increase in parallel speed-up with the addition of more processors to the parallel execution

segmentation

Process of partitioning a digital image into multiple segments. It is typically used to locate objects and boundaries in images

serialisation

In the context of data storage and transmission, is the process of converting a data structure or object state into a format that can be stored or transmitted across the network connection link. It is also called deflating or marshalling. C++ does not provide direct support for serialization

shared memory

Describes a computer architecture where all processors have direct access to common physical memory. It can directly address and access the same logical memory locations regardless of where the physical memory actually exists

slave

Process that receives tasks from the master process, executes them and returns the results back to the master process. It runs on a node called "slave node"

space filling

Curves whose range contains the entire 2D unit square or more generally, a N–dimensional hypercube. Because Giuseppe Peano (1858–1932) was the first to discover one, space-filling curves in the 2-dimensional plane are commonly called Peano curves

speed-up

Simplest and most widely used indicator for measuring the performance of a parallel program. It is defined as the "wall-clock time of serial execution" over the "wall–clock time of parallel execution"

starvation

Multitasking-related problem, where a process is perpetually denied necessary resources. Without those resources, the program can never finish its tasks

sub-graph

Graph whose vertex set is a subset of it and whose adjacency relation is a subset of it, restricted to this subset

synchronisation

The coordination of parallel tasks in real time. Often implemented by establishing a synchronization point within an application where a task may not proceed further until another task(s) reaches the same or logically equivalent point. Synchronization usually

involves waiting by at least one task, and can therefore cause a parallel application's wall clock execution time to increase

task

Logically discrete section of computational work, typically a program or program-like set of instructions that is executed by a processor. A parallel program consists of multiple tasks running on multiple processors

template

Programming technique that allows functions and classes to operate with generic types

tessellation

Pattern of plane figures that fills the plane with no overlaps and no gaps. Generalizations to higher dimensions are also possible

topology

Layout pattern of interconnections of various elements, such as computers of a network, components of a computer or vertices and edges of a graph

torus

In geometry, is a surface of revolution generated by revolving a circle in three dimensional space about an axis coplanar with the circle

tree

Undirected graph in which any two vertices are connected by exactly one simple path, that is, a graph without cycles

unmarshalling

See deserialisation

vertices

Fundamental unit out of which graphs are formed. Called also node

walk

Finite sequence of vertices and edges of a graph such that each edge in the sequence takes off where the last one left

Index

VI

APPENDICES

A

Message Passing Interface (MPI)

The following list includes the basic MPI primitives, cited in the present work. Not all of them have been used in the proposed solution:

MPI_Allgather: operation that gathers data from all processes and distributes it to all processes.

MPI_Allgatherv: operation that gathers data from all processes and delivers it to all. Each process may contribute a different amount of data.

MPI_Allreduce: operation that combines values from all processes and distributes the result back to all processes.

MPI_Alltoall: operation where all processes send data to all processes.

MPI_Alltoallv: operation where all processes send different amount of data to, and receive different amount of data from, all processes.

MPI_Barrier: Blocks until all processes have reached this routine.

MPI_Bcast: operation that broadcasts a message from the process with rank root to all other processes of the group.

MPI_Comm_rank: operation that determines the rank of the calling process in the communicator.

MPI_Comm_size: operation that returns the size of the group associated with a communicator.

MPI_Finalize: operation that terminates the MPI execution environment.

MPI_Gather: operation that gathers values from a group of processes.

MPI_Gatherv: operation that gathers varying amounts of data from all processes to the root process

MPI_Init: operation that initializes the MPI execution environment

MPI_Iprobe: operation that test for a message in a non-blocking way.

MPI_Irecv: operation that starts a standard-mode, non-blocking receive.

MPI_Isend: operation that starts a standard-mode, non-blocking send.

MPI_Probe: operation that allow checking of incoming messages, without actual receipt of them. The user can then decide how to receive them, based on the information returned by the probe in the status variable. For example, the user may allocate memory for the receive buffer, according to the length of the probed message.

MPI_Recv: operation that performs a standard-mode blocking receive.

MPI_Reduce: operation that reduces values on all processes within a group.

MPI_Reduce_scatter: operation that combines values and scatters the results.

MPI_Scan: operation that computes an inclusive scan (partial reduction).

MPI_Scatter: operation that sends data from one task to all tasks in a group.

MPI_Scatterv: operation that scatters a buffer in parts to all tasks in a group.

MPI_Send: operation that performs a standard-mode blocking send.

MPI_Wait: operation that waits for an MPI send or receive to complete.

B

PROTOBUFFERS

The information presented in this first section is a copy of the manual written by the Google staff, accessible in their developer manual and website.

B.1 What are protocol buffers?

Protocol buffers are a flexible, efficient, automated mechanism for serializing structured data — think XML (Extensible Markup Language), but smaller, faster, and simpler. You define how you want your data to be structured once, then you can use special generated source code to easily write and read your structured data to and from a variety of data streams and using a variety of languages. You can even update your data structure without breaking deployed programs that are compiled against the “old” format.

B.2 How do they work?

You specify how you want the information you're serializing to be structured by defining protocol buffer message types in .proto files. Each protocol buffer message is a small logical record of information, containing a series of name–value pairs. Here's a very basic example of a .proto file that defines a message containing information about a person:

```
message Person
{
  required string name = 1;
  required int32 id = 2;
  optional string email = 3;

  enum PhoneType
  {
    MOBILE = 0;
    HOME = 1;
    WORK = 2;
  }

  message PhoneNumber
  {
    required string number = 1;
    optional PhoneType type = 2 [default = HOME];
  }

  repeated PhoneNumber phone = 4;
}
```

As you can see, the message format is simple — each message type has one or more uniquely numbered fields, and each field has a name and a value type, where value types can be numbers (integer or floating–point), booleans, strings, raw bytes, or even (as in the example above) other protocol buffer message types, allowing you to structure your data hierarchically. You can specify optional fields, required fields, and repeated fields.

Once you've defined your messages, you run the protocol buffer compiler for your application's language on your .proto file to generate data access classes. These provide simple accessors for each field (like query() and set_query()) as well as methods to serialize/parse the whole structure to/from raw bytes — so, for instance, if your chosen language is C++, running the compiler on the above example will generate a class called Person. You can then use this class in your application to populate, serialize, and retrieve Person protocol buffer messages. You might then write some code like this:

```
Person person;
person.set_name("John Doe");
person.set_id(1234);
person.set_email("jdoe@example.com");
fstream output("myfile", ios::out | ios::binary);
person.SerializeToOstream(&output);
```

Then, later on, you could read your message back in:

```
fstream input("myfile", ios::in | ios::binary);
Person person;
person.ParseFromIstream(&input);
cout << "Name: " << person.name() << endl;
cout << "E-mail: " << person.email() << endl;
```

You can add new fields to your message formats without breaking backwards-compatibility; old binaries simply ignore the new field when parsing. So if you have a communications protocol that uses protocol buffers as its data format, you can extend your protocol without having to worry about breaking existing code.

Why not just use XML?

Protocol buffers have many advantages over XML for serializing structured data. Protocol buffers:

- are simpler.
- are 3 to 10 times smaller.
- are 20 to 100 times faster.
- are less ambiguous.
- generate data access classes that are easier to use programmatically.

For example, let's say you want to model a person with a name and an email. In XML, you need to do:

```
<person>
    <name>John Doe</name>
    <email>jdoe@example.com</email>
</person>
```

while the corresponding protocol buffer message (in protocol buffer text format) is:

```
# Textual representation of a protocol buffer.
# This is *not* the binary format used on the wire.
person
{
    name: "John Doe"
    email: "jdoe@example.com"
}
```

When this message is encoded to the protocol buffer binary format (the text format above is just a convenient human-readable representation for debugging and editing), it would probably be 28 bytes long and take around 100-200 nanoseconds to parse. The XML version is at least 69 bytes if you remove whitespace, and would take around 5000-10000 nanoseconds to parse.

Also, manipulating a protocol buffer is much easier:

```
cout << "Name: " << person.name() << endl;
cout << "E-mail: " << person.email() << endl;
```

Whereas with XML you would have to do something like:

```
cout << "Name: "
     << person.getElementsByTagName("name")->item(0)->innerText()
     << endl;
cout << "E-mail: "
     << person.getElementsByTagName("email")->item(0)->innerText()
     << endl;
```

However, protocol buffers are not always a better solution than XML — for instance, protocol buffers would not be a good way to model a text-based document with markup (e.g. HTML), since you cannot easily interleave structure with text. In addition, XML is human-readable and human-editable; protocol buffers, at least in their native format, are not. XML is also — to some extent — self-describing. A protocol buffer is only meaningful if you have the message definition (the .proto file).

B.3 Message definitions

The framework defines a set of messages in four .proto files which defines the named configuration files used by the user to specify how the framework behaves for a certain problem. These .proto files are:

`confcluster.proto:` message definitions for the cluster configuration file.

`confloadbalancer.proto:` message definitions for the load balancer configuration file.

`confpartitioner.proto:` message definitions for the domain decomposition configuration file.

`confproblem.proto:` message definitions for the problem configuration file.

The equivalent .cpp and .h automatic generated code files implement the *Configuration* namespaces

Cluster configuration .proto file

The `confcluster.proto` file defines the messages used to define the structures of the cluster configuration file, that is, the structures that the user has to specify in order to configure the cluster used to run the serial parallelised application.

A cluster structure is nothing else than a set of machines (message *Node*) and connections between them (message *Link*).

```
package Configurator;

message Node
{
    required string name =1;
    optional double cpu =2;
    optional double memory =3;
}

message Link
{
    required string source =1;
    required string destination =2;
    optional double weight =3;
}

message Cluster
{
    repeated Node node =1;
    repeated Link link =2;
}
```

A *Node* is formed by the hostname and its CPU and memory capacity features, while a

Link is defined by the source and target hostfile and the weight of the link, that is, a measure of the velicity of the link (related for example on a ICMP (Internet Control Message Protocol) reply time).

Load balancer configuration .proto file

The `confloadbalancer.proto` file defines the messages used to define the structures of the load balancer configuration file, that is, the structures that the user has to specify in order to configure the load balancer used to run the serial parallelised application.

The load balancer structure consists of a set of repeated plugins , each one described by its name (to being able to load the corresponding dynamic library) and the weight provided to the global load balanced scenario. Optionally, a historical component can be defined, to take into account characteristics of previous executions.

```
package Configurator;

message Plugin
{
    optional string name =1;
    optional double weight =2;
}

message Historical
{
    repeated double weight =1;
}

message LoadBalancer
{
    repeated Plugin plugin =1;
    optional Historical historical =2;
}
```

Domain decomposer configuration .proto file

The `confpartitioner.proto` file defines the messages used to define the structures of the partitioner configuration file, that is, the structures that the user has to specify in order to configure the domain decomposer used to run the serial parallelised application.

The main structure is called *Partitioner* and it is defined by two parameters:

Heuristic: that provides the domain decomposition update rules.

Method: that provides the namely method to decompose the problem into partitions.

To define the `method` structure, it is required to specify both the type of partition (static domain or dynamic domain decomposition), that is, whether the partition shape can change while the execution of the algorithm (refer to page 103), and also the name of the method

```
package Configurator;

message Constant
{
    required int32 steps =1;
}

message Lineal
{
    required int32 initial =1;
    required int32 increment =2;
}

message Annealing
{
    required int32 initial =1;
    required double temperature =2;
    required int32 increment =3;
}

message PID
{
    required int32 initial =1;
    required double P =2;
    required double I =3;
    required double D =4;
}

message Heuristic
{
    optional Constant constant =1;
    optional Lineal lineal =2;
    optional Annealing annealing =3;
    optional PID pid =4;
}

message Method
{
    enum Type
    {
        STATIC =1;
        DYNAMIC =2;
    }

    required Type type =1;
    required string name =2;
}

message Partitioner
{
    required Heuristic heuristic =1;
    required Method method =2;
}
```

to use to decompose the domain of the problem. This name will be concatenated with the prefix of the folder where the plugins are and the file extension of a dynamic library.

The `heuristic` structure rules when the loadbalancer has to be executed, depending on how have varied the parameters returned by the load balancer. It can be defined by at least one of the following methods:

Constant: it is executed once the algorithm has completed a certain number of steps. This steps are delayed a constant number of steps.

Lineal: like the `Constant` one but, in this case, the interval between two consecutive update steps varies depending on a fixed increment value.

Annealing: the steps, when the partitioner algorithm has to be updated, follows a simulation annealing probabilistic method.

PID: the updating method is treated as a system that, once stabilised and taking as input the values of the load balancing returned values, gives the steps in which the partitioner algorithm has to be executed.

Problem configuration .proto file

The `confproblem.proto` file defines the messages used to define the structures of the problem configuration file, that is, the structures that the user has to specify in order to configure the application that runs on the cluster.

The main structure is called *Simulation* and it is defined by five parameters:

name: used as pattern for the filename of the output files, logs,

cells: basic tessellation that defines the lattice. At the moment, there are three tessellations implemented: triangles, squares and hexagons (see figure 1.3).

topology: set of connections of the application graph and dimension of the lattice. The connections can be specified node by node (graph topology) or specified globally for all the nodes of the application graph (matrix topology).

input: specification of the input values file with its relative path.

output: specification of the format of the output files, such as .txt files or graphic file formats. Each output format is specified in a output plugin but more than a file can be used to visualise the output files.

```
package Configurator;

message Coordinate
{
    required string name =1;
    repeated int32 distance =2;
}

message SquareCell
{
    repeated Coordinate coordinate =1;
}

message TriangularCell
{
    repeated Coordinate coordinate =1;
}

message HexagonalCell
{
    repeated Coordinate coordinate =1;
}

message Cell
{
    optional SquareCell square =1;
    optional TriangularCell triangular =2;
    optional HexagonalCell hexagonal =3;
}

message Connection
{
    optional int32 source =1;
    required string direction =2;
}

message Matrix
{
    repeated bool boundary =1;
    repeated Connection all =2;
}

message Graph
{
    repeated bool boundary =1;
    repeated Connection link =2;
}

message Topology
{
    required int32 dimension =1;
    repeated int32 dimensions =2;
    optional Matrix matrix =3;
    optional Graph graph =4;
}

message Input
{
    optional string inputFile =1 [default = "stdin"];
}
```

```
message OutputFormat
{
    required string name =1;
    required string type =2;
}

message Output
{
    enum How
    {
        GLOBAL =1;
        LOCAL =2;
    }

    optional string timeFile =1 [default = "stdout"];
    optional string logFile =2 [default = "stdout"];
    optional How how =3 [default = GLOBAL];
    optional bool partitions =4 [default = false];
    repeated OutputFormat file =5;
}

message Simulation
{
    required string name =1;
    required Cell cells =2;
    required Topology topology =3;
    required Input input =4;
    required Output output =5;
}
```

C

TESTS SCENARIO

The platform has been tested in several scenarios, although in the results section of this document are shown only two of them: *hlrb-II* cluster and *metis*.

	hlrb-II	**metis**
Main use	Performance tests	Functionalities tests
Cores	9728	2
Peak performance	62.3 TFlop/s	4.4 GFlop/s
Size of memory	39 TByte	4 GByte
Memory per core	4 GByte	2 GByte
Processor type	Intel Itanium2 Montecito Dual Core	AMD Opteron
Clock rate	1.6 GHz	2.2 GHz
Bandwidth	6.4 GByte/s	1 GByte/s

(a) Hardware features of hlrb-II and metis machines

	hlrb-II	**metis**
Operating System	SLES 10 SP3	Gentoo
kernel	2.6.16	2.6.31-gentoo-r6
libc version	2.4	2.13-r1
gcc version	4.1.2	4.5.2
MPI version	2.2	2.3
Protobuffers version	2.2.0	2.4.0
Scalasca version	1.3.1	1.3.3

(b) Software information of hlrb-II and metis machines

Table C.1: Hardware and Software features of the machines used to run the platform

D

PUBLICATIONS

List of contributions published as articles, conference proceedings:

- LAFLANG'11: "Parallel simulation of NEPs on clusters" (not yet published), IEEE, C.B.Navarrete, M.Cruz, J.M.Rojas, E.Anguiano, A.Ortega.
- PARA'10: "Framework for the execution of sequential code in a loadbalanced parallel scenario" (not yet published), LNCS, C.B.Navarrete, E.Anguiano and M.Gerndt.
- ISPC'08: "Parallel Metropolis-Montecarlo simulation for Potts model using an adaptable network topology based on dynamic graph partitioning", pages 89–96, ISBN:978-0-7695-3472-5, C.Castañeda-Marroquín, C.B.Navarrete, A.Ortega, M.Alfosenca and E.Anguiano.
- PARA'08: "Efficient parallel implementation of Potts model" (not yet published), LNCS, C.B.Navarrete, C.Castañeda-Marroquín, A.Ortega, M.Alfosenca and E.Anguiano.
- ICPDC'07: "Reconfiguration of the topology of farms of heterogeneous computers using an architecture of dynamic graph partitioning", pages 589–590 (1), ISBN: 978-988-98671-5-7, IAENG, C.B.Navarrete, E.Anguiano.
- PDCS2007: "Architecture based on dynamic graphs for the dynamic reconfiguration of farms of computers", pages 425–429, ISBN: 978-975-01752-0-6, WASET, C.B.Navarrete, E.Anguiano.
- PARELEC'06: "MPI and non-MPI simulations for epitaxial surface growth", ISBN:0-7695-2554-74-447, IEEE, C.B.Navarrete, S.Holgado, E.Anguiano.
- PARA'06: "Epitaxial surface growth with local interaction, parallel and non-parallel simulations", ISBN: 978-3-540-75754-22-889, LNCS, C.B.Navarrete, S.Holgado. E.Anguiano.

Chapter published in books:

- High Performance Computing on Vector Systems 2009, Springer, ISBN: 978-3-642-03912-6, Chapter III: Applications, Editors: M.Resch, S.Roller, K.Benkert, M.Galle, W.Bez, H.Kobayashi. Authors: E.Anguiano and C.B.Navarrete. November 2009
- Science and Supercomputing in Europe, Annual Report, ISBN: 978-88-86037-22-8, Pages: 242–248. C.B.Navarrete. July 2009

Universidad Autónoma de Madrid
Escuela Politécnica Superior

Thesis to obtain PhD. degree in
Computer Science and
Telecommunication Engineering
by Universidad Autónoma de Madrid

Thesis advisor: Dr. Eloy Anguiano Rey

Platform for automatic paralellisation of sequential codes using dynamic graphs partitioning and based on user-adaptable load balancing

Carmen Blanca Navarrete Navarrete

Reference Manual

This thesis was presented on 2011
Tribunal:

Dr. Michael Gerndt
Dr. José María Carazo
Dr. Jesús María González Barahona
Dr. Iván González Martínez
Dr. Alfonso Ortega de la Puente

Carmen Blanca Navarrete Navarrete
Platform for automatic paralellisation of sequential codes using dynamic graphs partitioning and based on user-adaptable load balancing

Carmen Blanca Navarrete Navarrete
Escuela Politécnica Superior. Dpto Ingeniería Informática

IMPRESO EN ESPAÑA – PRINTED IN SPAIN

INDEX

1

Todo List

Member Reconfiguration::Process::Process(int, Algorithm ∗, int) Create a communicator channel with my master process.

Create a communicator channel with my master process.

2 Namespace Index

2.1 Namespace List

Here is a list of all namespaces with brief descriptions:

3 Class Index

3.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

4 Class Index

4.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

5 File Index

5.1 File List

Here is a list of all files with brief descriptions:

6 Namespace Documentation

6.1 Configurator Namespace Reference

6.1.1 Namespaces

- namespace cc

6.1.2 Classes

- struct StaticDescriptorInitializer_confcluster_2eproto
- class Node
- class Link
- class Cluster
- struct StaticDescriptorInitializer_confloadbalancer_2eproto
- class Plugin
- class Historical
- class LoadBalancer
- struct StaticDescriptorInitializer_confpartitioner_2eproto
- class Constant
- class Lineal
- class Annealing
- class PID
- class Kalman
- class Heuristic
- class Method

- class Partitioner
- struct StaticDescriptorInitializer_confproblem_2eproto
- class Coordinate
- class SquareCell
- class TriangularCell
- class HexagonalCell
- class Cell
- class Connection
- class Matrix
- class Graph
- class Topology
- class OutputFormat
- class Input
- class Output
- class Simulation

6.1.3 Enumerations

- enum Method_Type { Method_Type_STATIC = 1, Method_Type_DYNAMIC = 2 }
- enum Output_How { Output_How_GLOBAL = 1, Output_How_LOCAL = 2 }

6.1.4 Functions

- void protobuf_AssignDesc_confcluster_2eproto ()
- void protobuf_ShutdownFile_confcluster_2eproto ()
- void protobuf_AddDesc_confcluster_2eproto ()
- void protobuf_AssignDesc_confloadbalancer_2eproto ()
- void protobuf_ShutdownFile_confloadbalancer_2eproto ()
- void protobuf_AddDesc_confloadbalancer_2eproto ()
- void protobuf_AssignDesc_confpartitioner_2eproto ()
- void protobuf_ShutdownFile_confpartitioner_2eproto ()
- void protobuf_AddDesc_confpartitioner_2eproto ()
- const ::google::protobuf::EnumDescriptor * Method_Type_descriptor ()

- bool Method_Type_IsValid (int value)
- const ::std::string & Method_Type_Name (Method_Type value)
- bool Method_Type_Parse (const ::std::string &name, Method_Type *value)
- void protobuf_AssignDesc_confproblem_2eproto ()
- void protobuf_ShutdownFile_confproblem_2eproto ()
- void protobuf_AddDesc_confproblem_2eproto ()
- const ::google::protobuf::EnumDescriptor * Output_How_descriptor ()
- bool Output_How_IsValid (int value)
- const ::std::string & Output_How_Name (Output_How value)
- bool Output_How_Parse (const ::std::string &name, Output_How *value)

6.1.5 Variables

- struct Configurator::StaticDescriptorInitializer_confcluster_2eproto static_descriptor_initializer_confcluster_2eproto_
- struct Configurator::StaticDescriptorInitializer_confloadbalancer_2eproto static_descriptor_initializer_confloadbalancer_2eproto_
- struct Configurator::StaticDescriptorInitializer_confpartitioner_2eproto static_descriptor_initializer_confpartitioner_2eproto_
- const Method_Type Method_Type_Type_MIN = Method_Type_STATIC
- const Method_Type Method_Type_Type_MAX = Method_Type_DYNAMIC
- const int Method_Type_Type_ARRAYSIZE = Method_Type_Type_MAX + 1
- struct Configurator::StaticDescriptorInitializer_confproblem_2eproto static_descriptor_initializer_confproblem_2eproto_
- const Output_How Output_How_How_MIN = Output_How_GLOBAL
- const Output_How Output_How_How_MAX = Output_How_LOCAL
- const int Output_How_How_ARRAYSIZE = Output_How_How_MAX + 1

6.1.6 Enumeration Type Documentation

enum Configurator::Method_Type

Enumerator:

Method_Type_STATIC

Method_Type_DYNAMIC

enum Configurator::Output_How

Enumerator:

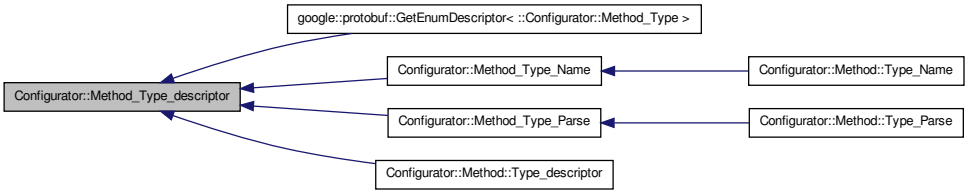
Output_How_GLOBAL

Output_How_LOCAL

6.1.7 Function Documentation

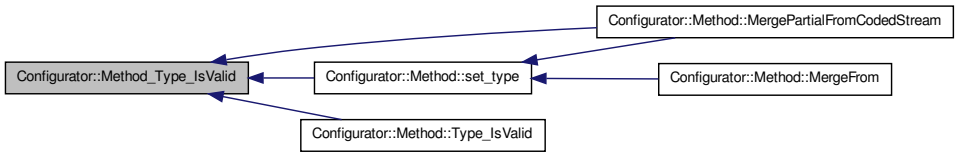
const ::google::protobuf::EnumDescriptor * Configurator::Method_Type_descriptor ( )

Here is the caller graph for this function:



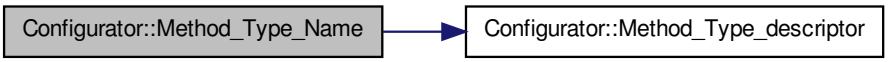
bool Configurator::Method_Type_IsValid ([int]value)

Here is the caller graph for this function:



const ::std::string& Configurator::Method_Type_Name ([Method_Type]value) `[inline]`

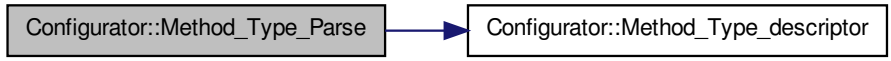
Here is the call graph for this function:



Here is the caller graph for this function:

bool Configurator::Method_Type_Parse ([const ::std::string &]name, Method_Type * *value*) [inline]

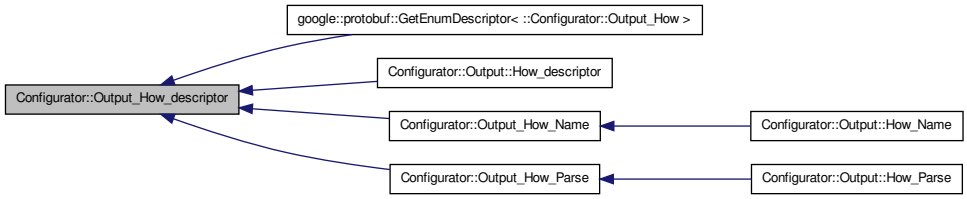
Here is the call graph for this function:



Here is the caller graph for this function:

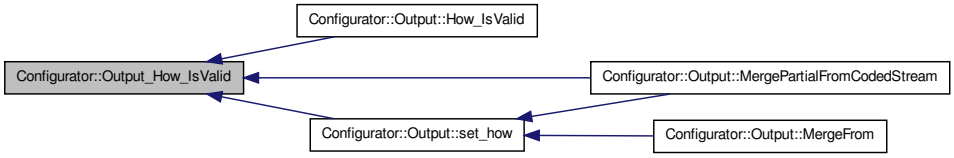
const ::google::protobuf::EnumDescriptor * Configurator::Output_How_descriptor ( )

Here is the caller graph for this function:



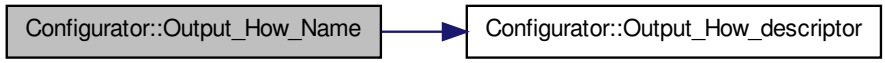
bool Configurator::Output_How_IsValid ([int]value)

Here is the caller graph for this function:



const ::std::string& Configurator::Output_How_Name ([Output_How]value) [inline]

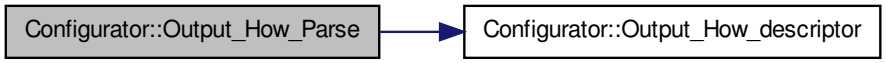
Here is the call graph for this function:



Here is the caller graph for this function:

bool Configurator::Output_How_Parse ([const ::std::string &]name, Output_How * *value*) [inline]

Here is the call graph for this function:

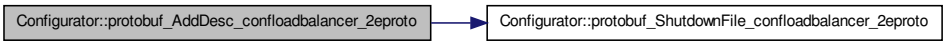


Here is the caller graph for this function:

void Configurator::protobuf_AddDesc_confcluster_2eproto ( )

void Configurator::protobuf_AddDesc_confloadbalancer_2eproto ( )

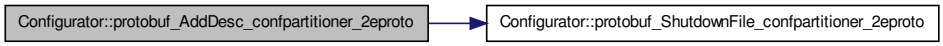
Here is the call graph for this function:



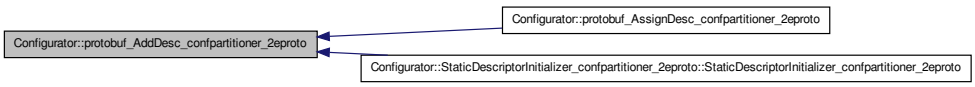
Here is the caller graph for this function:

void Configurator::protobuf_AddDesc_confpartitioner_2eproto ( )

Here is the call graph for this function:

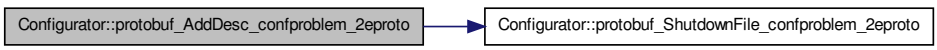


Here is the caller graph for this function:



void Configurator::protobuf_AddDesc_confproblem_2eproto ( )

Here is the call graph for this function:



Here is the caller graph for this function:

void Configurator::protobuf_AssignDesc_confcluster_2eproto ( )

void Configurator::protobuf_AssignDesc_confloadbalancer_2eproto ( )

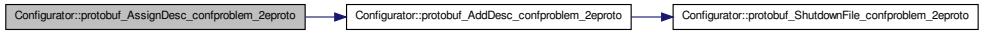
Here is the call graph for this function:

void Configurator::protobuf_AssignDesc_confpartitioner_2eproto ( )

Here is the call graph for this function:

void Configurator::protobuf_AssignDesc_confproblem_2eproto ( )

Here is the call graph for this function:



void Configurator::protobuf_ShutdownFile_confcluster_2eproto ( )

void Configurator::protobuf_ShutdownFile_confloadbalancer_2eproto ( )

Here is the caller graph for this function:

void Configurator::protobuf_ShutdownFile_confpartitioner_2eproto ( )

Here is the caller graph for this function:

void Configurator::protobuf_ShutdownFile_confproblem_2eproto ( )

Here is the caller graph for this function:

6.1.8 Variable Documentation

const int Configurator::Method_Type_Type_ARRAYSIZE = Method_Type_Type_MAX + 1

const Method_Type Configurator::Method_Type_Type_MAX = Method_Type_DYNAMIC

const Method_Type Configurator::Method_Type_Type_MIN = Method_Type_STATIC

const int Configurator::Output_How_How_ARRAYSIZE = Output_How_How_MAX + 1

const Output_How Configurator::Output_How_How_MAX = Output_How_LOCAL

const Output_How Configurator::Output_How_How_MIN = Output_How_GLOBAL

struct Configurator::StaticDescriptorInitializer_confcluster_2eproto
Configurator::static_descriptor_initializer_confcluster_2eproto_

struct Configurator::StaticDescriptorInitializer_confloadbalancer_2eproto
Configurator::static_descriptor_initializer_confloadbalancer_2eproto_

struct Configurator::StaticDescriptorInitializer_confpartitioner_2eproto
Configurator::static_descriptor_initializer_confpartitioner_2eproto_

struct Configurator::StaticDescriptorInitializer_confproblem_2eproto
Configurator::static_descriptor_initializer_confproblem_2eproto_

6.2 Configurator::cc Namespace Reference

6.2.1 Variables

- const ::google::protobuf::Descriptor * Node_descriptor_ = NULL
- const ::google::protobuf::internal::GeneratedMessageReflection * Node_reflection_ = NULL
- const ::google::protobuf::Descriptor * Link_descriptor_ = NULL
- const ::google::protobuf::internal::GeneratedMessageReflection * Link_reflection_ = NULL
- const ::google::protobuf::Descriptor * Cluster_descriptor_ = NULL
- const ::google::protobuf::internal::GeneratedMessageReflection * Cluster_reflection_ = NULL

6.2.2 Variable Documentation

const ::google::protobuf::Descriptor* Configurator::cc::Cluster_descriptor_ = NULL

const ::google::protobuf::internal::GeneratedMessageReflection* Configurator::cc::Cluster_reflection_ = NULL

const ::google::protobuf::Descriptor* Configurator::cc::Link_descriptor_ = NULL

const ::google::protobuf::internal::GeneratedMessageReflection* Configurator::cc::Link_reflection_ = NULL

const ::google::protobuf::Descriptor* Configurator::cc::Node_descriptor_ = NULL

const ::google::protobuf::internal::GeneratedMessageReflection* Configurator::cc::Node_reflection_ = NULL

6.3 Configure Namespace Reference

6.3.1 Classes

- class Configuration
- class ConfCluster
- class ConfLoadBalancer
- class ConfPartitioner
- class ConfProblem

6.4 google Namespace Reference

6.4.1 Namespaces

- namespace protobuf

6.5 google::protobuf Namespace Reference

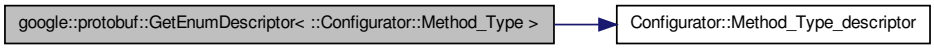
6.5.1 Functions

- template<>
  const EnumDescriptor * GetEnumDescriptor< ::Configurator::Method_Type > ()
- template<>
  const EnumDescriptor * GetEnumDescriptor< ::Configurator::Output_How > ()

6.5.2 Function Documentation

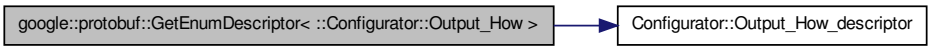
template<> const EnumDescriptor* google::protobuf::GetEnumDescriptor< ::Configurator::Method_Type > ( ) [inline]

Here is the call graph for this function:



template<> const EnumDescriptor* google::protobuf::GetEnumDescriptor< ::Configurator::Output_How > ( ) [inline]

Here is the call graph for this function:



6.6 Reconfiguration Namespace Reference

6.6.1 Classes

- class Algorithm
- class Communicator
- class Coordinate
- class Data
- class baseException
- class OutOfBoundsException
- class InvalidArgumentsException
- class NoSuchFileException
- class ConfigurationException
- class InternalException
- class MPIException
- class Framework
- class Machine
- class Link
- class Vertex
- class Graph
- class IFormat
- class Item
- class Queue
- class IGrouping
- class IPlugin
- class ISerializable
- class ISorting
- class Min
- class Output
- class MapElements
- class Partition
- class Problem
- class Process

- class SuperMaster
- class Slave
- class RMS
- class String
- class Timer
- class Vector

6.6.2 Typedefs

- typedef IFormat * formatcreate_t ()
- typedef void formatdestroy_t (IFormat *)
- typedef IGrouping * groupincreate_t ()
- typedef void groupingdestroy_t (IGrouping *)
- typedef IPlugin * plugincreate_t ()
- typedef void plugindestroy_t (IPlugin *)
- typedef ISorting * sortcreate_t ()
- typedef void sortdestroy_t (ISorting *)

6.6.3 Typedef Documentation

IFormat * Reconfiguration::formatcreate_t

Needed to call the dynamic library.

Reconfiguration::formatdestroy_t

Need to free the dynamic library.

IGrouping * Reconfiguration::groupincreate_t

Needed to call the dynamic library.

Reconfiguration::groupingdestroy_t

Need to free the dynamic library.

IPlugin ∗ Reconfiguration::plugincreate_t

Needed to call the dynamic library.

Reconfiguration::plugindestroy_t

Need to free the dynamic library.

ISorting ∗ Reconfiguration::sortcreate_t

Needed to call the dynamic library.

Reconfiguration::sortdestroy_t

Need to free the dynamic library.

7 Class Documentation

7.1 Reconfiguration::Algorithm Class Reference

`#include <Algorithm.h>`

7.1.1 Public Member Functions

- Algorithm ()

  Constructor of the abstract class Algorithm.

- virtual ∼Algorithm ()

  virtual destructor of the class Algorithm.

- virtual int process (const char ∗, Data ∗∗, int)=0

  Reads the input file and pass the processed values to the middleware.

- virtual void kernel (Graph ∗∗)=0

  Executes the user-defined code.

- virtual void save (Graph ∗, std::stringstream ∗)=0

  Pass to the middleware the information that must be saved as an image or as a text file.

- virtual int end (Graph ∗)=0

  Checks whether the simulation has to be ended.

- void set (int)

Sets the rank of the specific slave to the iRank variable of the algorithm object.

- virtual void postGather (Graph **)

  Executes a global operation over the global problem data values.

7.1.2 Public Attributes

- int iRank

7.1.3 Detailed Description

This layer is needed to pass the user Algorithm variables to the kernel of the middleware.

7.1.4 Constructor & Destructor Documentation

Reconfiguration::Algorithm::Algorithm ( )

Constructor of the abstract class Algorithm.

Returns

*this

Creates an object Algorithm.

Reconfiguration::Algorithm::~Algorithm ( ) `[virtual]`

virtual destructor of the class Algorithm.

Destructor of the class Algorithm.

Returns

void

Destroys the Algorithm object.

7.1.5 Member Function Documentation

virtual int Reconfiguration::Algorithm::end ([Graph *]) [pure virtual]

Checks whether the simulation has to be ended.

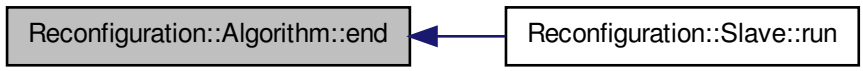
Parameters

in	*data*	Pointer to the graph of values.

Returns

int 1 if the end condition evaluation returns true, 0 otherwise. Evaluates the user condition to check if the simulation ends. Method implemented by the user

Here is the caller graph for this function:



virtual void Reconfiguration::Algorithm::kernel ([Graph **]) [pure virtual]

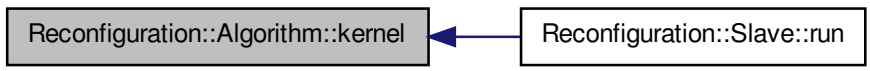
Executes the user-defined code.

Returns

void

All the information needed to execute the kernel user algorithm is allocated in the graph. The graph is a collection of Data datatype user defined values. Method implemented by the user

Here is the caller graph for this function:



void Reconfiguration::Algorithm::postGather ([Graph **]graph) [virtual]

Executes a global operation over the global problem data values.

Parameters

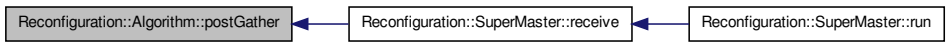
in	*graph*	Application graph pointer.

Returns

void

Executes a global operation over the global problem data values, when the gather has been performed. By default this function returns directly without performing any operation but can be overwritten by the user.

Here is the caller graph for this function:



virtual int Reconfiguration::Algorithm::process ([const char *], Data ** , int) [pure virtual]

Reads the input file and pass the processed values to the middleware.

Returns

int value specifying the error code. Reads the values of the user-specifyed input file. Method implemented by the user

Here is the caller graph for this function:

virtual void Reconfiguration::Algorithm::save ([Graph *], std::stringstream *) [pure virtual]

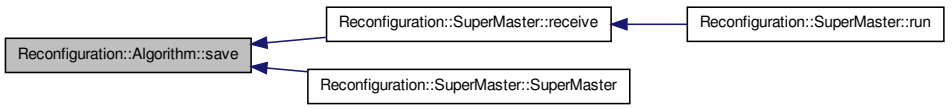
Pass to the middleware the information that must be saved as an image or as a text file.

Returns

void

This function is really a customized-dump. Method implemented by the user

Here is the caller graph for this function:



void Reconfiguration::Algorithm::set ([int]iRank)

Sets the rank of the specific slave to the iRank variable of the algorithm object.

Sets the rank of the specific slave to the rank variable of the algorithm object.

Parameters

in	*iRank*	Identifier of the slave.

Returns

void

Method needed to let the kernel of the middleware to know the rank identifier of the slave

Parameters

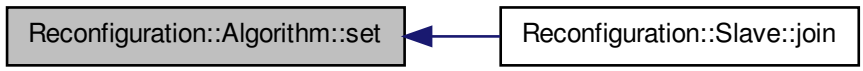
in	*iRank*	Identifier of the slave.

Returns

void

Method needed to let the kernel of the middleware to know the rank ID of the slave

Here is the caller graph for this function:



7.1.6 Member Data Documentation

int Reconfiguration::Algorithm::iRank

MPI identifier of the actual process.

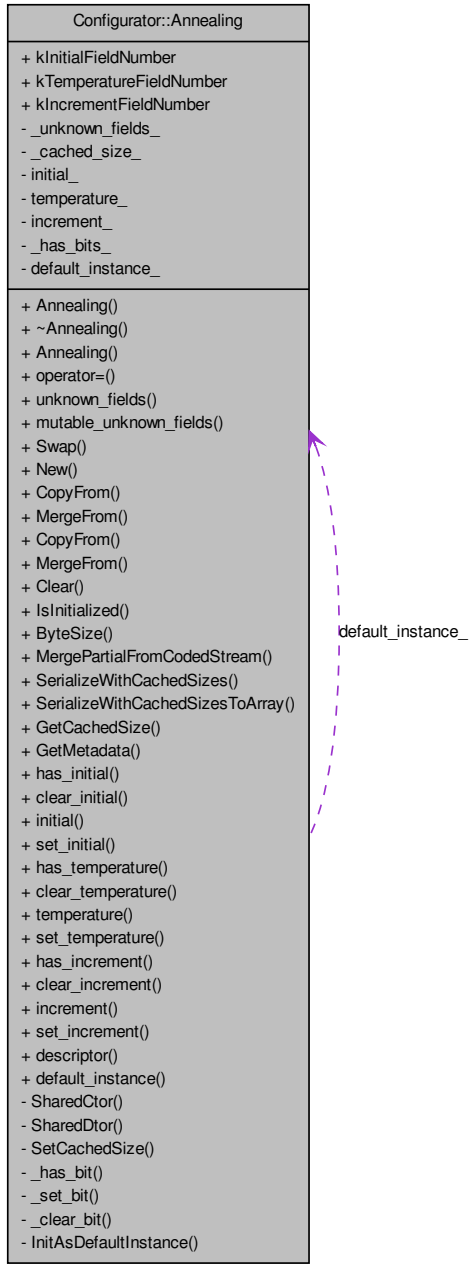
The documentation for this class was generated from the following files:

- Algorithm.h
- Algorithm.cpp

7.2 Configurator::Annealing Class Reference

`#include <confpartitioner.pb.h>`

Collaboration diagram for Configurator::Annealing:



7.2.1 Public Member Functions

- Annealing ()
- virtual ∼Annealing ()
- Annealing (const Annealing &from)
- Annealing & operator= (const Annealing &from)
- const ::google::protobuf::UnknownFieldSet & unknown_fields () const
- inline::google::protobuf::UnknownFieldSet ∗ mutable_unknown_fields ()
- void Swap (Annealing ∗other)
- Annealing ∗ New () const
- void CopyFrom (const ::google::protobuf::Message &from)
- void MergeFrom (const ::google::protobuf::Message &from)
- void CopyFrom (const Annealing &from)
- void MergeFrom (const Annealing &from)
- void Clear ()
- bool IsInitialized () const
- int ByteSize () const
- bool MergePartialFromCodedStream (::google::protobuf::io::CodedInputStream ∗input)
- void SerializeWithCachedSizes (::google::protobuf::io::CodedOutputStream ∗output) const
- ::google::protobuf::uint8 ∗ SerializeWithCachedSizesToArray (::google::protobuf::uint8 ∗output) const
- int GetCachedSize () const
- ::google::protobuf::Metadata GetMetadata () const
- bool has_initial () const
- void clear_initial ()
- inline::google::protobuf::int32 initial () const
- void set_initial (::google::protobuf::int32 value)
- bool has_temperature () const
- void clear_temperature ()
- double temperature () const
- void set_temperature (double value)
- bool has_increment () const
- void clear_increment ()

- inline::google::protobuf::int32 increment () const
- void set_increment (::google::protobuf::int32 value)

7.2.2 Static Public Member Functions

- static const ::google::protobuf::Descriptor * descriptor ()
- static const Annealing & default_instance ()

7.2.3 Static Public Attributes

- static const int kInitialFieldNumber = 1
- static const int kTemperatureFieldNumber = 2
- static const int kIncrementFieldNumber = 3

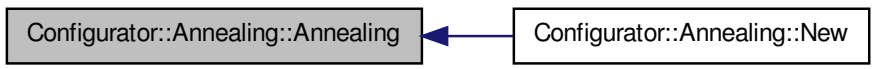
7.2.4 Friends

- void protobuf_AddDesc_confpartitioner_2eproto ()
- void protobuf_AssignDesc_confpartitioner_2eproto ()
- void protobuf_ShutdownFile_confpartitioner_2eproto ()

7.2.5 Constructor & Destructor Documentation

Configurator::Annealing::Annealing ( )

Here is the caller graph for this function:



Configurator::Annealing::~Annealing ( ) `[virtual]`

Configurator::Annealing::Annealing ([const Annealing &]from)

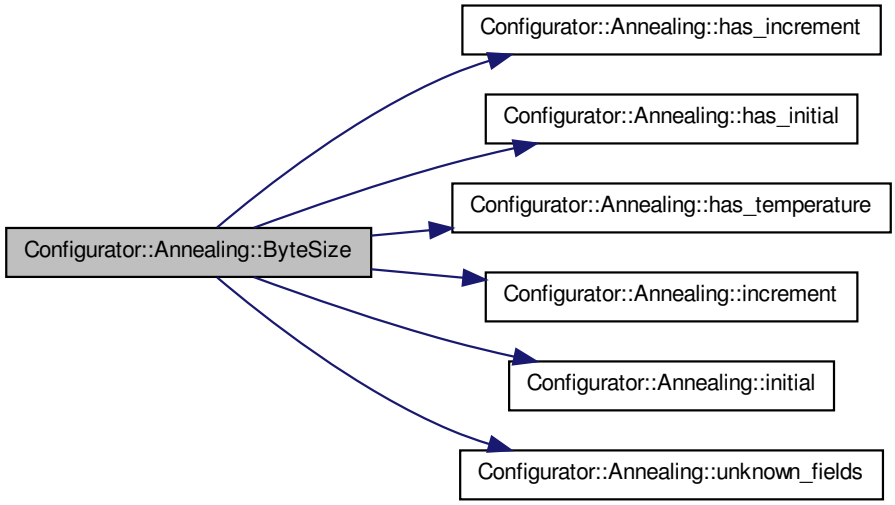
Here is the call graph for this function:


Configurator::Annealing::Annealing
Configurator::Annealing::MergeFrom

7.2.6 Member Function Documentation

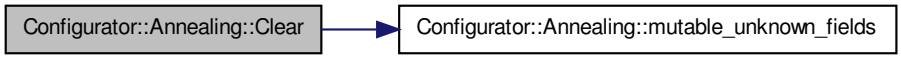
int Configurator::Annealing::ByteSize ( ) const

Here is the call graph for this function:

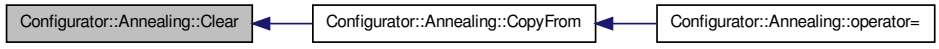


void Configurator::Annealing::Clear ( )

Here is the call graph for this function:



Here is the caller graph for this function:



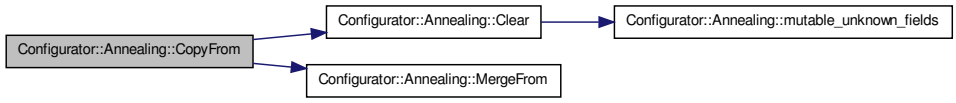
void Configurator::Annealing::clear_increment ( ) [inline]

void Configurator::Annealing::clear_initial ( ) [inline]

void Configurator::Annealing::clear_temperature ( ) [inline]

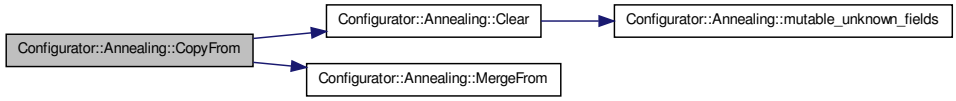
void Configurator::Annealing::CopyFrom ([const Annealing &]from)

Here is the call graph for this function:



void Configurator::Annealing::CopyFrom ([const ::google::protobuf::Message &]from)

Here is the call graph for this function:



Here is the caller graph for this function:

const Annealing & Configurator::Annealing::default_instance ( ) [static]

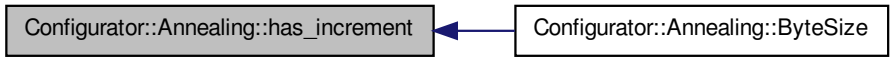
const ::google::protobuf::Descriptor * Configurator::Annealing::descriptor ( ) [static]

int Configurator::Annealing::GetCachedSize ( ) const [inline]

google::protobuf::Metadata Configurator::Annealing::GetMetadata ( ) const

bool Configurator::Annealing::has_increment ( ) const [inline]

Here is the caller graph for this function:

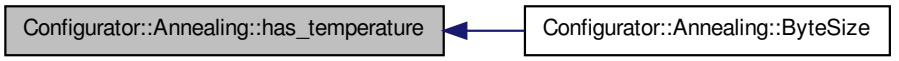


bool Configurator::Annealing::has_initial ( ) const [inline]

Here is the caller graph for this function:

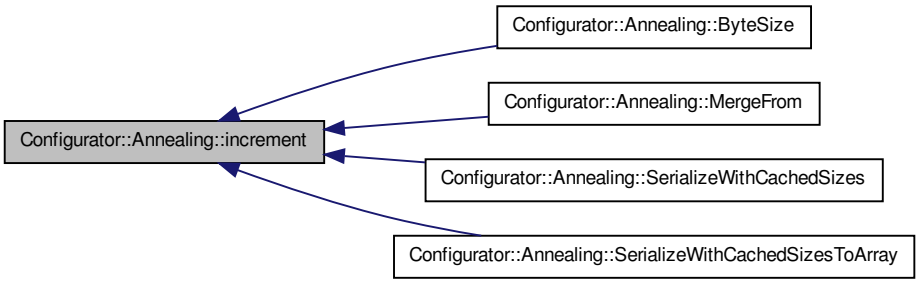
bool Configurator::Annealing::has_temperature ( ) const [inline]

Here is the caller graph for this function:



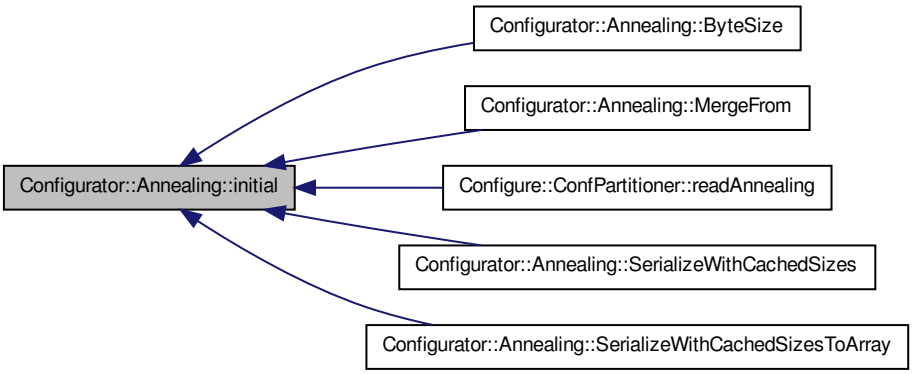
google::protobuf::int32 Configurator::Annealing::increment ( ) const [inline]

Here is the caller graph for this function:



google::protobuf::int32 Configurator::Annealing::initial ( ) const [inline]

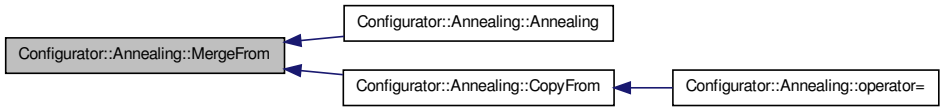
Here is the caller graph for this function:



bool Configurator::Annealing::IsInitialized ( ) const

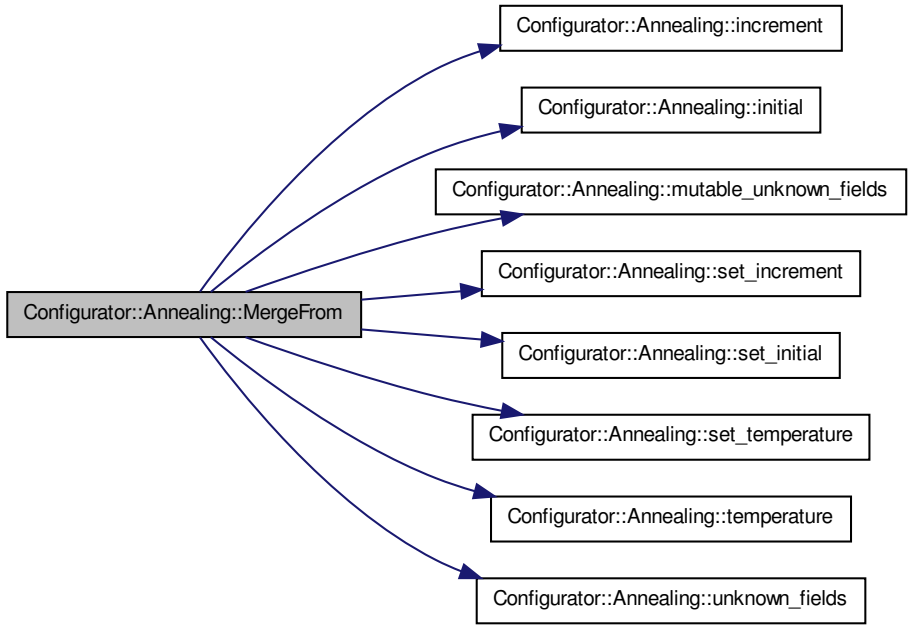
void Configurator::Annealing::MergeFrom ([const ::google::protobuf::Message &]from)

Here is the caller graph for this function:



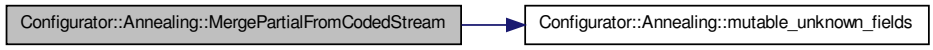
void Configurator::Annealing::MergeFrom ([const Annealing &]from)

Here is the call graph for this function:



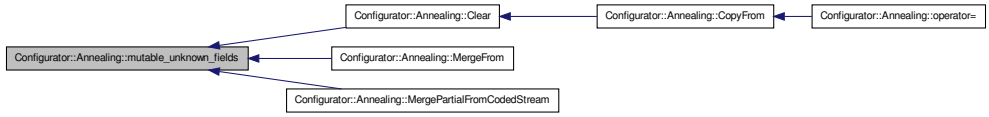
bool Configurator::Annealing::MergePartialFromCodedStream ([::google::protobuf::io::CodedInputStream ∗]input)

Here is the call graph for this function:



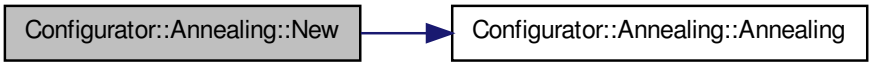
inline ::google::protobuf::UnknownFieldSet∗ Configurator::Annealing::mutable_-unknown_fields ( ) [inline]

Here is the caller graph for this function:



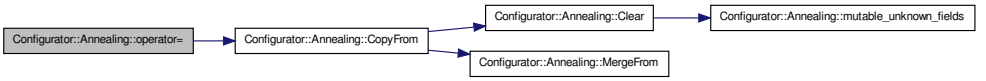
Annealing * Configurator::Annealing::New ( ) const

Here is the call graph for this function:



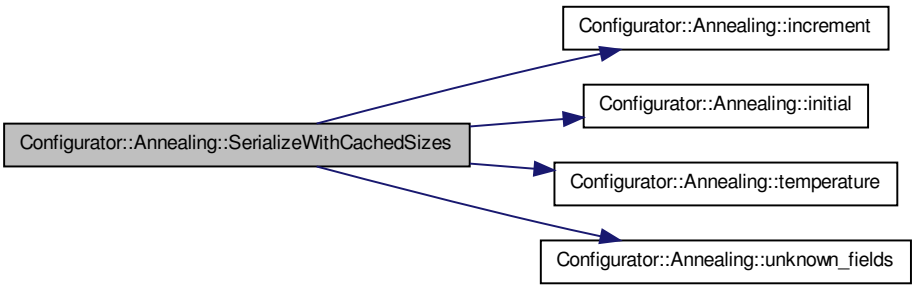
Annealing& Configurator::Annealing::operator= ([const Annealing &]from) `[inline]`

Here is the call graph for this function:



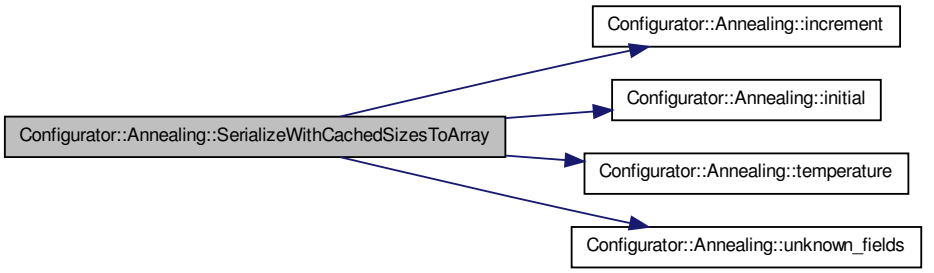
void Configurator::Annealing::SerializeWithCachedSizes ([::google::protobuf::io::CodedOutputStream *]output) const

Here is the call graph for this function:



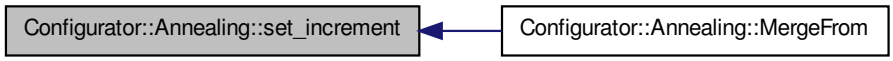
google::protobuf::uint8 * Configurator::Annealing::SerializeWithCachedSizesToArray ([::google::protobuf::uint8 *]output) const

Here is the call graph for this function:



void Configurator::Annealing::set_increment ([::google::protobuf::int32]value) [inline]

Here is the caller graph for this function:

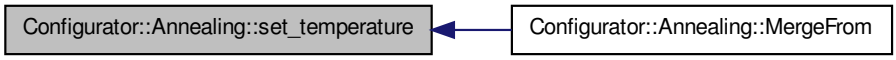


void Configurator::Annealing::set_initial ([::google::protobuf::int32]value) [inline]

Here is the caller graph for this function:

void Configurator::Annealing::set_temperature ([double]value) [inline]

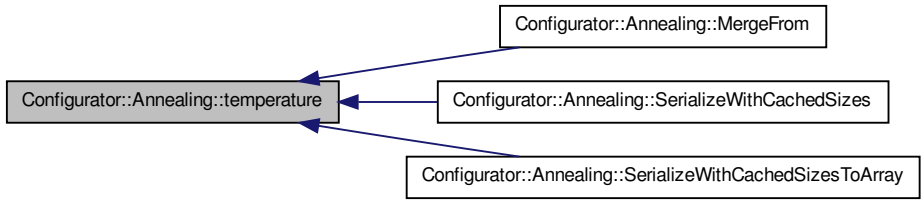
Here is the caller graph for this function:



void Configurator::Annealing::Swap ([Annealing ∗]other)

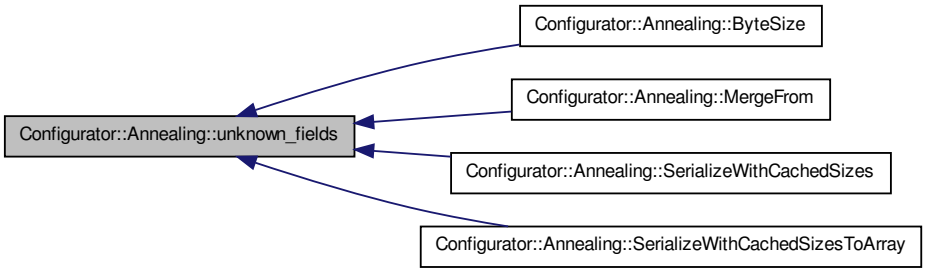
double Configurator::Annealing::temperature ( ) const [inline]

Here is the caller graph for this function:



const ::google::protobuf::UnknownFieldSet& Configurator::Annealing::unknown_-fields ( ) const [inline]

Here is the caller graph for this function:



7.2.7 Friends And Related Function Documentation

void protobuf_AddDesc_confpartitioner_2eproto ( ) [friend]

void protobuf_AssignDesc_confpartitioner_2eproto ( ) [friend]

void protobuf_ShutdownFile_confpartitioner_2eproto ( ) [friend]

7.2.8 Member Data Documentation

const int Configurator::Annealing::kIncrementFieldNumber = 3 [static]

const int Configurator::Annealing::kInitialFieldNumber = 1 [static]

const int Configurator::Annealing::kTemperatureFieldNumber = 2 [static]

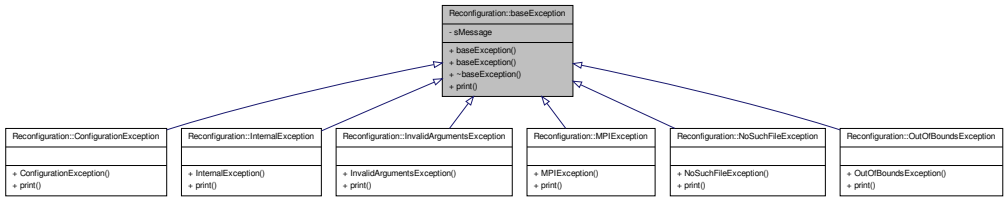
The documentation for this class was generated from the following files:

- confpartitioner.pb.h
- confpartitioner.pb.cc

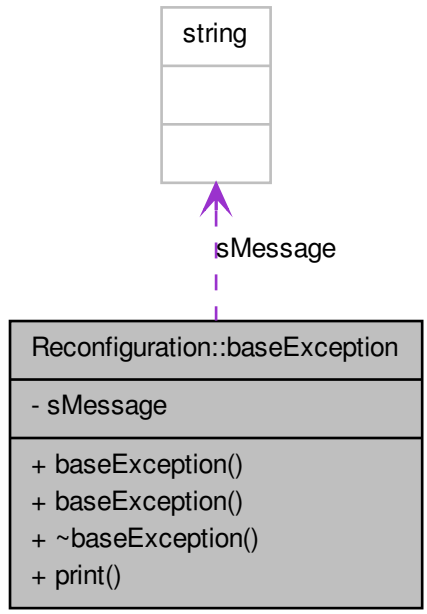
7.3 Reconfiguration::baseException Class Reference

`#include <Exception.h>`

Inheritance diagram for Reconfiguration::baseException:



Collaboration diagram for Reconfiguration::baseException:



7.3.1 Public Member Functions

- baseException ()

  Constructor of the baseException class.

- baseException (std::string)

  Constructor of the baseException class.

- virtual ~baseException ()

  virtual destructor of the class baseException.

- void print ()

  Prints the exception.

7.3.2 Detailed Description

base Exception father of all the customized exceptions that can be thrown during the execution of the framework.

7.3.3 Constructor & Destructor Documentation

Reconfiguration::baseException::baseException ( )

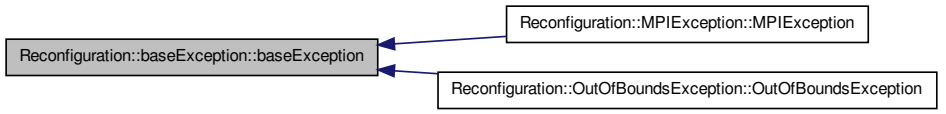
Constructor of the baseException class.

Returns

∗this

Creates an object baseException.

Here is the caller graph for this function:



Reconfiguration::baseException::baseException ([std::string]sText)

Constructor of the baseException class.

Parameters

in	*sText*	Exception string.

Returns

∗this

Creates an object baseException and sets the message of the exception.

Reconfiguration::baseException::~baseException ( ) [virtual]

virtual destructor of the class baseException.

Returns

void

Destroys the baseException object.

7.3.4 Member Function Documentation

void Reconfiguration::baseException::print ( )

Prints the exception.

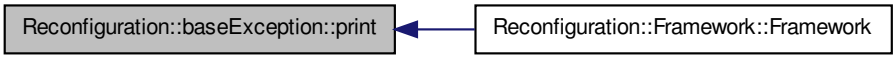
Returns

void

Prints the exception by printing its message.

Reimplemented in Reconfiguration::OutOfBoundsException, Reconfiguration::InvalidArgumentsEx Reconfiguration::NoSuchFileException, Reconfiguration::ConfigurationException, Reconfiguration::InternalException, and Reconfiguration::MPIException.

Here is the caller graph for this function:



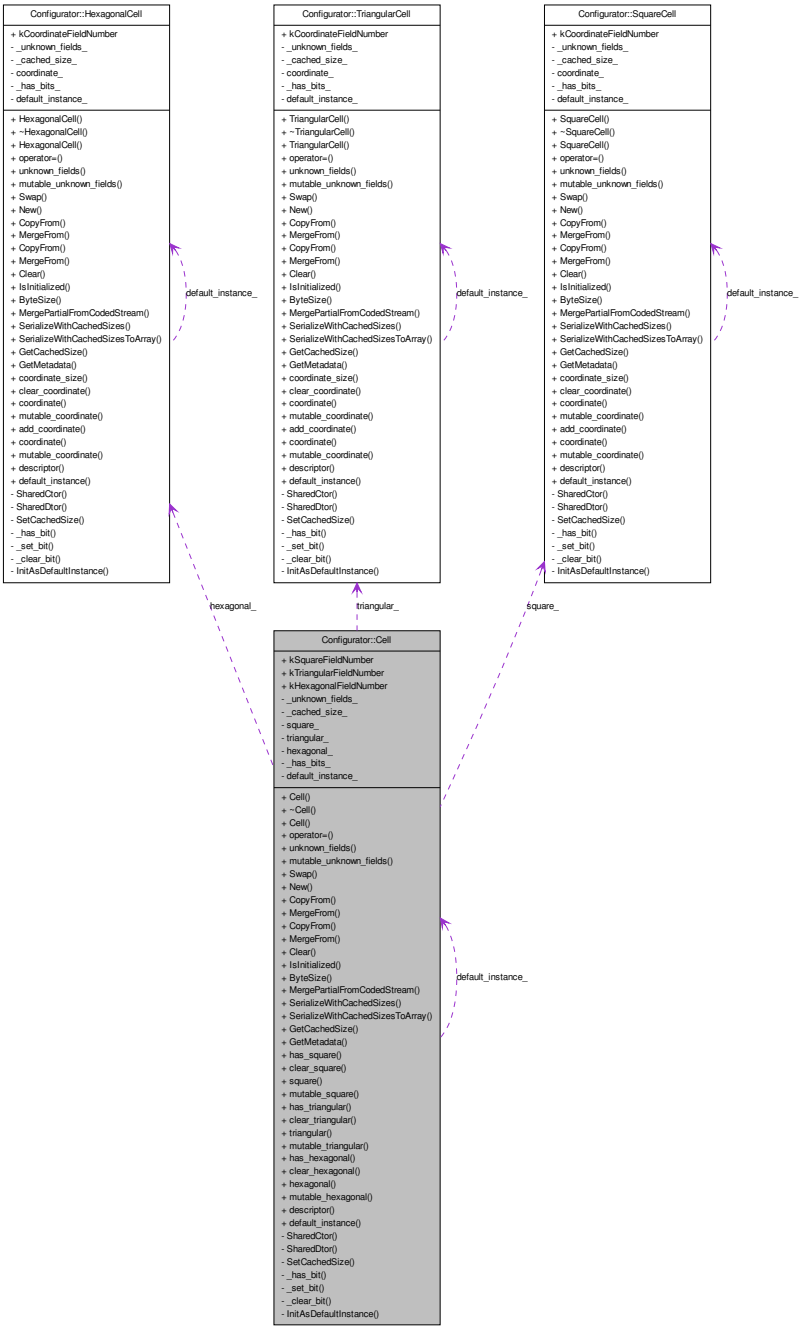
The documentation for this class was generated from the following files:

- Exception.h
- Exception.cpp

7.4 Configurator::Cell Class Reference

`#include <confproblem.pb.h>`

Collaboration diagram for Configurator::Cell:



7.4.1 Public Member Functions

- Cell ()
- virtual ∼Cell ()
- Cell (const Cell &from)
- Cell & operator= (const Cell &from)
- const ::google::protobuf::UnknownFieldSet & unknown_fields () const
- inline::google::protobuf::UnknownFieldSet ∗ mutable_unknown_fields ()
- void Swap (Cell ∗other)
- Cell ∗ New () const
- void CopyFrom (const ::google::protobuf::Message &from)
- void MergeFrom (const ::google::protobuf::Message &from)
- void CopyFrom (const Cell &from)
- void MergeFrom (const Cell &from)
- void Clear ()
- bool IsInitialized () const
- int ByteSize () const
- bool MergePartialFromCodedStream (::google::protobuf::io::CodedInputStream ∗input)
- void SerializeWithCachedSizes (::google::protobuf::io::CodedOutputStream ∗output) const
- ::google::protobuf::uint8 ∗ SerializeWithCachedSizesToArray (::google::protobuf::uint8 ∗output) const
- int GetCachedSize () const
- ::google::protobuf::Metadata GetMetadata () const
- bool has_square () const
- void clear_square ()
- const ::Configurator::SquareCell & square () const
- inline::Configurator::SquareCell ∗ mutable_square ()
- bool has_triangular () const
- void clear_triangular ()
- const ::Configurator::TriangularCell & triangular () const
- inline::Configurator::TriangularCell ∗ mutable_triangular ()
- bool has_hexagonal () const
- void clear_hexagonal ()

- const ::Configurator::HexagonalCell & hexagonal () const
- inline::Configurator::HexagonalCell ∗ mutable_hexagonal ()

7.4.2 Static Public Member Functions

- static const ::google::protobuf::Descriptor ∗ descriptor ()
- static const Cell & default_instance ()

7.4.3 Static Public Attributes

- static const int kSquareFieldNumber = 1
- static const int kTriangularFieldNumber = 2
- static const int kHexagonalFieldNumber = 3

7.4.4 Friends

- void protobuf_AddDesc_confproblem_2eproto ()
- void protobuf_AssignDesc_confproblem_2eproto ()
- void protobuf_ShutdownFile_confproblem_2eproto ()

7.4.5 Constructor & Destructor Documentation

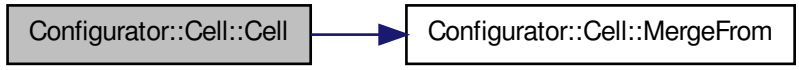
Configurator::Cell::Cell ( )

Here is the caller graph for this function:

Configurator::Cell::∼Cell ( ) `[virtual]`

Configurator::Cell::Cell ([const Cell &]from)

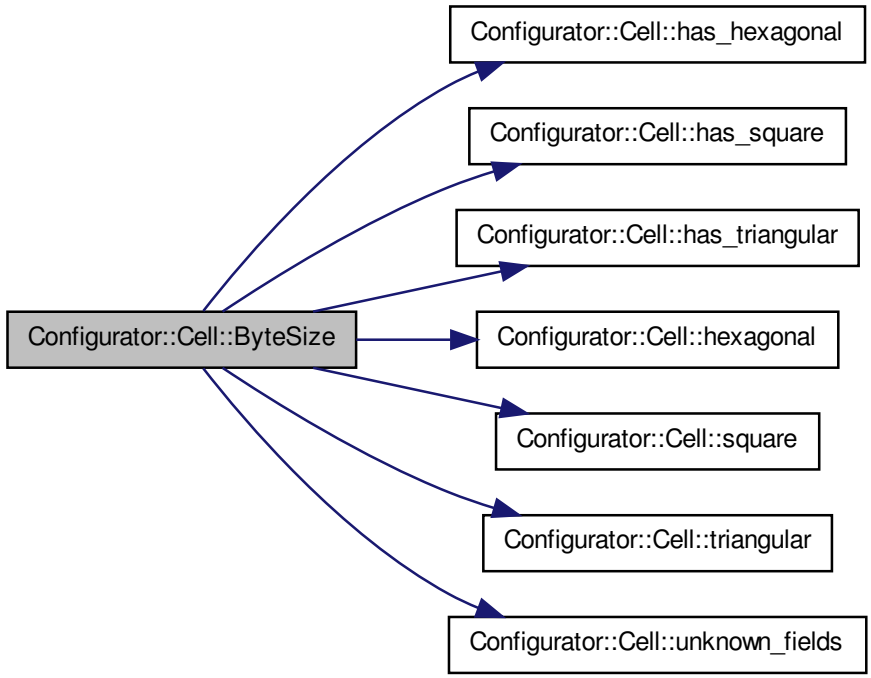
Here is the call graph for this function:



7.4.6 Member Function Documentation

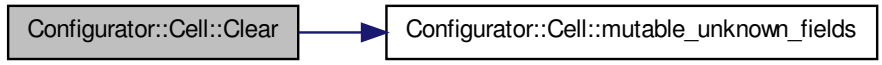
int Configurator::Cell::ByteSize ( ) const

Here is the call graph for this function:

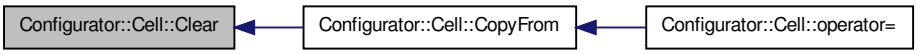


void Configurator::Cell::Clear ( )

Here is the call graph for this function:



Here is the caller graph for this function:



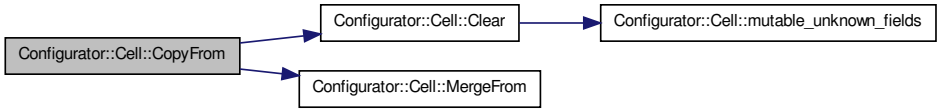
void Configurator::Cell::clear_hexagonal ( ) [inline]

void Configurator::Cell::clear_square ( ) [inline]

void Configurator::Cell::clear_triangular ( ) [inline]

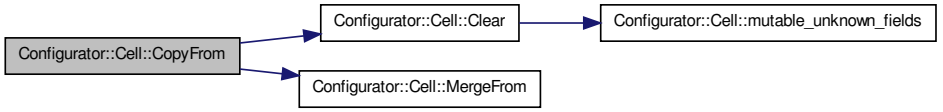
void Configurator::Cell::CopyFrom ([const Cell &]from)

Here is the call graph for this function:



void Configurator::Cell::CopyFrom ([const ::google::protobuf::Message &]from)

Here is the call graph for this function:



Here is the caller graph for this function:

const Cell & Configurator::Cell::default_instance ( ) [static]

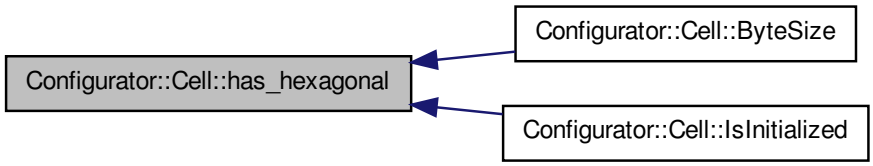
const ::google::protobuf::Descriptor * Configurator::Cell::descriptor ( ) [static]

int Configurator::Cell::GetCachedSize ( ) const [inline]

google::protobuf::Metadata Configurator::Cell::GetMetadata ( ) const

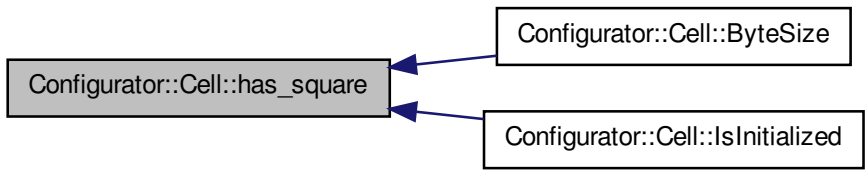
bool Configurator::Cell::has_hexagonal ( ) const [inline]

Here is the caller graph for this function:



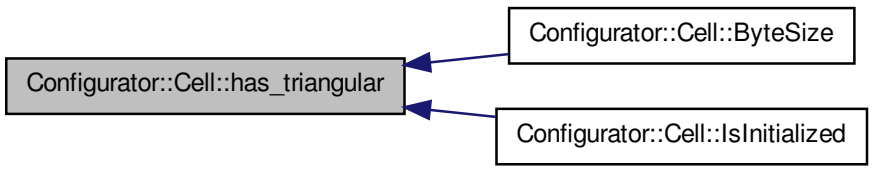
bool Configurator::Cell::has_square ( ) const [inline]

Here is the caller graph for this function:



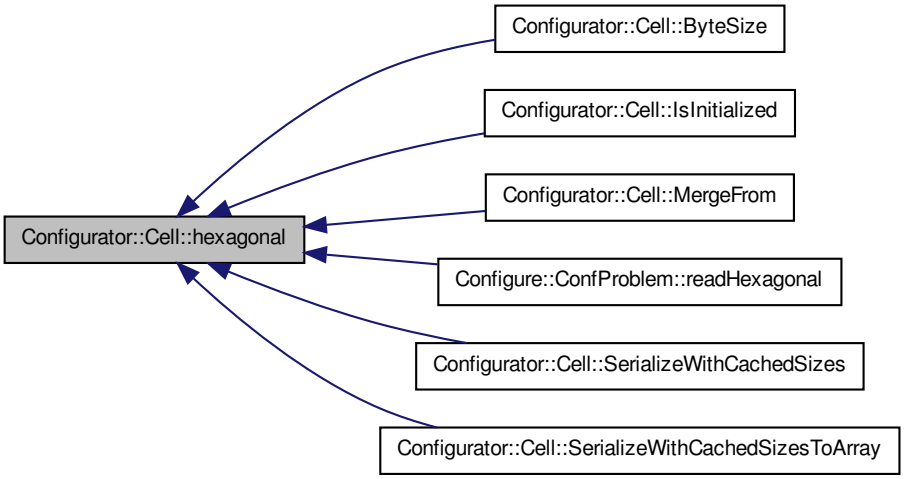
bool Configurator::Cell::has_triangular ( ) const [inline]

Here is the caller graph for this function:



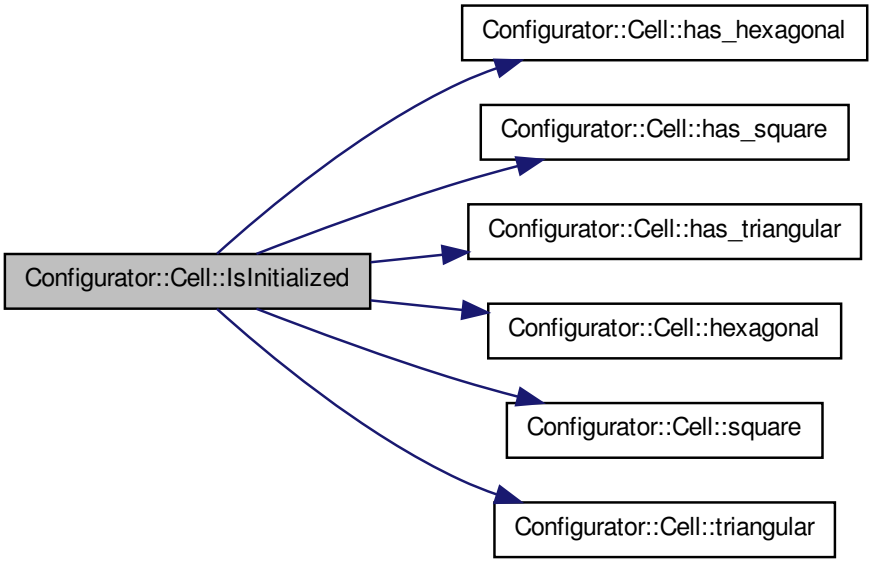
const ::Configurator::HexagonalCell & Configurator::Cell::hexagonal ( ) const [inline]

Here is the caller graph for this function:



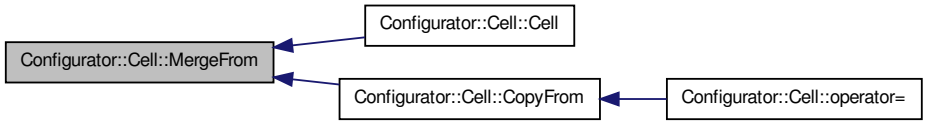
bool Configurator::Cell::IsInitialized ( ) const

Here is the call graph for this function:



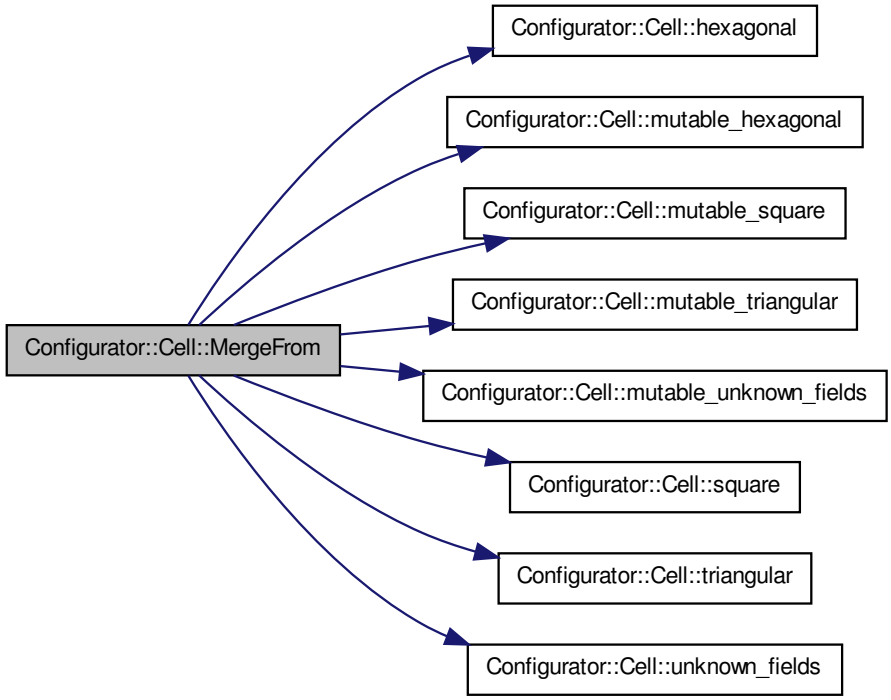
void Configurator::Cell::MergeFrom ([const ::google::protobuf::Message &]from)

Here is the caller graph for this function:



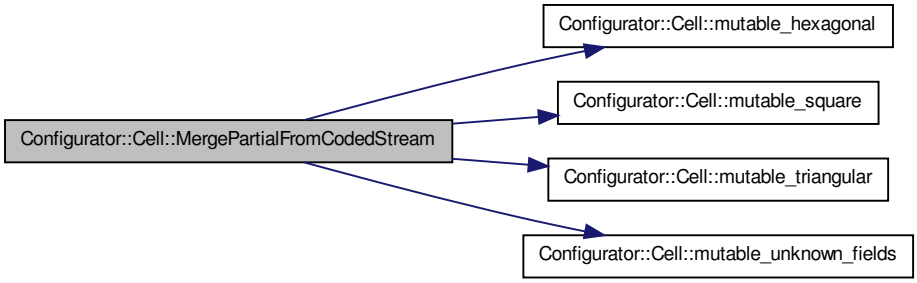
void Configurator::Cell::MergeFrom ([const Cell &]from)

Here is the call graph for this function:



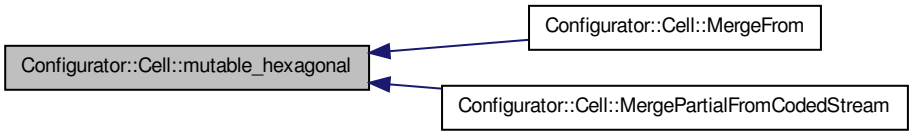
bool Configurator::Cell::MergePartialFromCodedStream ([::google::protobuf::io::CodedInputStream *]input)

Here is the call graph for this function:



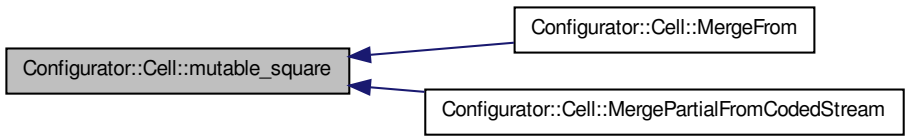
Configurator::HexagonalCell * Configurator::Cell::mutable_hexagonal ( ) [inline]

Here is the caller graph for this function:



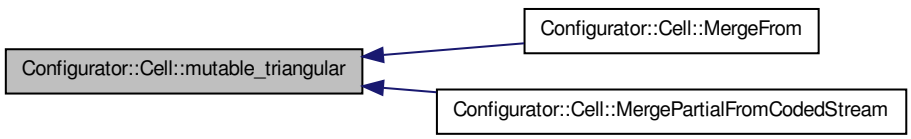
Configurator::SquareCell * Configurator::Cell::mutable_square ( ) [inline]

Here is the caller graph for this function:



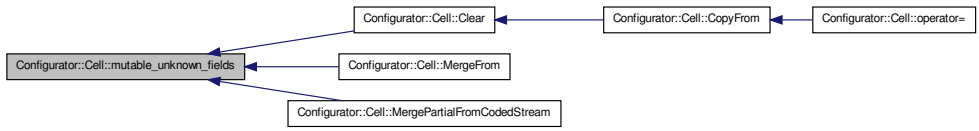
Configurator::TriangularCell * Configurator::Cell::mutable_triangular ( ) [inline]

Here is the caller graph for this function:



inline ::google::protobuf::UnknownFieldSet* Configurator::Cell::mutable_unknown_-fields ( ) [inline]

Here is the caller graph for this function:

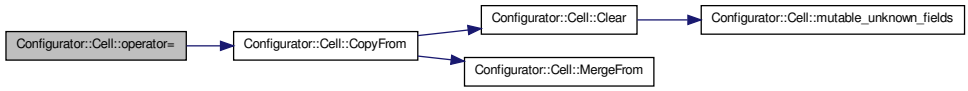


Cell * Configurator::Cell::New ( ) const

Here is the call graph for this function:

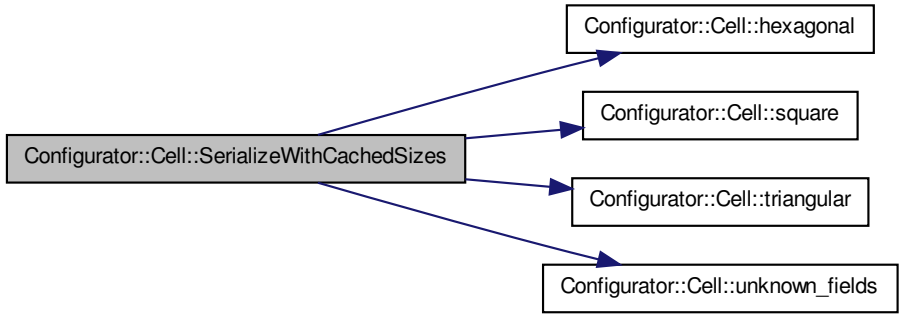
Cell& Configurator::Cell::operator= ([const Cell &]from) `[inline]`

Here is the call graph for this function:



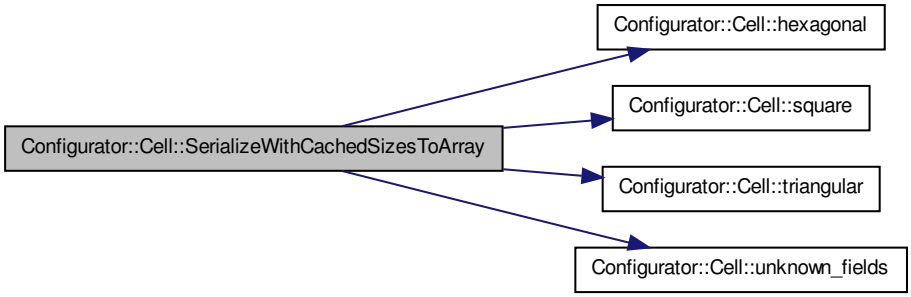
void Configurator::Cell::SerializeWithCachedSizes ([::google::protobuf::io::CodedOutputStream *]output) const

Here is the call graph for this function:



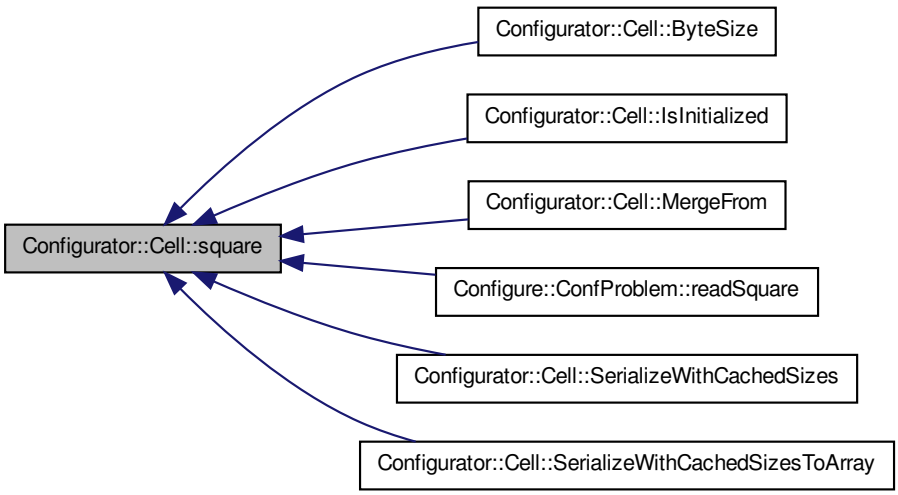
google::protobuf::uint8 * Configurator::Cell::SerializeWithCachedSizesToArray ([::google::protobuf::uint8 *]output) const

Here is the call graph for this function:



const ::Configurator::SquareCell & Configurator::Cell::square ( ) const [inline]

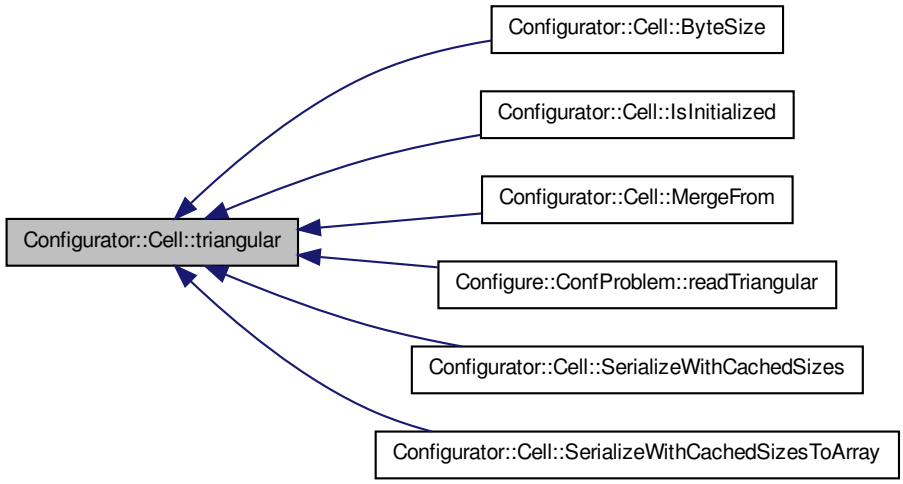
Here is the caller graph for this function:



void Configurator::Cell::Swap ([Cell *]other)

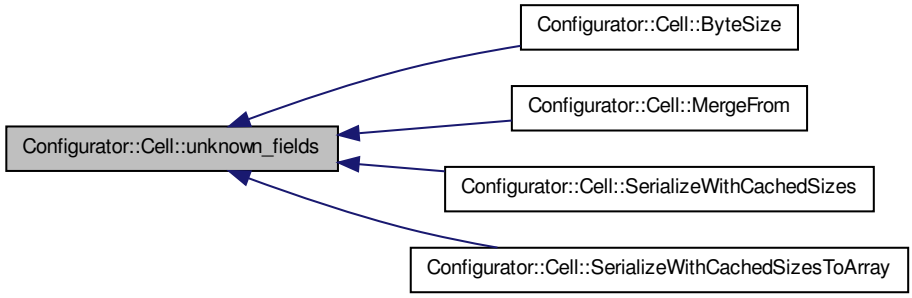
const ::Configurator::TriangularCell & Configurator::Cell::triangular ( ) const [inline]

Here is the caller graph for this function:



const ::google::protobuf::UnknownFieldSet& Configurator::Cell::unknown_fields ( ) const `[inline]`

Here is the caller graph for this function:



7.4.7 Friends And Related Function Documentation

void protobuf_AddDesc_confproblem_2eproto ( ) `[friend]`

void protobuf_AssignDesc_confproblem_2eproto ( ) `[friend]`

void protobuf_ShutdownFile_confproblem_2eproto ( ) `[friend]`

7.4.8 Member Data Documentation

const int Configurator::Cell::kHexagonalFieldNumber = 3 `[static]`

const int Configurator::Cell::kSquareFieldNumber = 1 `[static]`

const int Configurator::Cell::kTriangularFieldNumber = 2 `[static]`

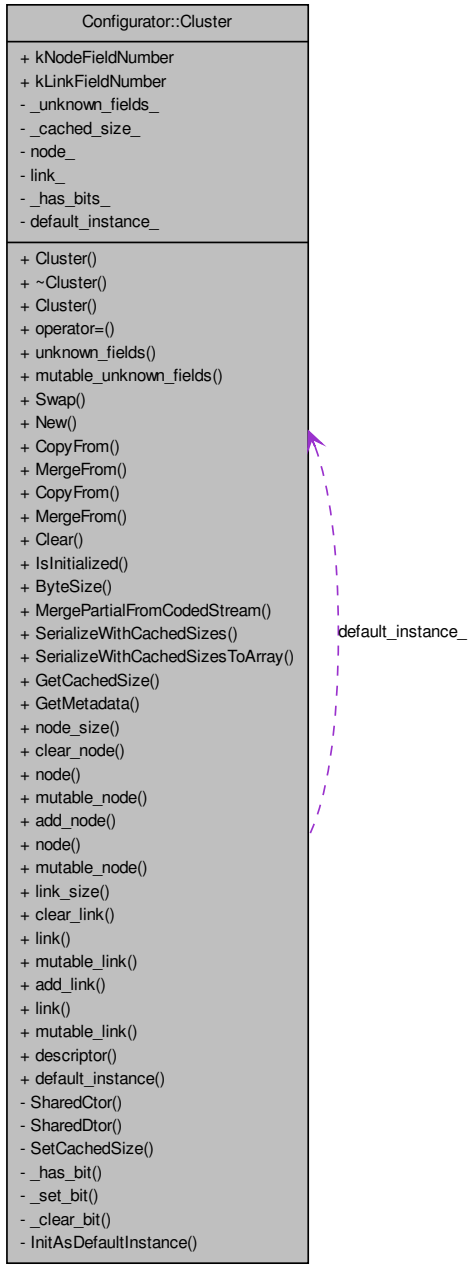
The documentation for this class was generated from the following files:

- confproblem.pb.h
- confproblem.pb.cc

7.5 Configurator::Cluster Class Reference

`#include <confcluster.pb.h>`

Collaboration diagram for Configurator::Cluster:



7.5.1 Public Member Functions

- Cluster ()
- virtual ∼Cluster ()
- Cluster (const Cluster &from)
- Cluster & operator= (const Cluster &from)
- const ::google::protobuf::UnknownFieldSet & unknown_fields () const
- inline::google::protobuf::UnknownFieldSet ∗ mutable_unknown_fields ()
- void Swap (Cluster ∗other)
- Cluster ∗ New () const
- void CopyFrom (const ::google::protobuf::Message &from)
- void MergeFrom (const ::google::protobuf::Message &from)
- void CopyFrom (const Cluster &from)
- void MergeFrom (const Cluster &from)
- void Clear ()
- bool IsInitialized () const
- int ByteSize () const
- bool MergePartialFromCodedStream (::google::protobuf::io::CodedInputStream ∗input)
- void SerializeWithCachedSizes (::google::protobuf::io::CodedOutputStream ∗output) const
- ::google::protobuf::uint8 ∗ SerializeWithCachedSizesToArray (::google::protobuf::uint8 ∗output) const
- int GetCachedSize () const
- ::google::protobuf::Metadata GetMetadata () const
- int node_size () const
- void clear_node ()
- const ::Configurator::Node & node (int index) const
- inline::Configurator::Node ∗ mutable_node (int index)
- inline::Configurator::Node ∗ add_node ()
- const ::google::protobuf::RepeatedPtrField< ::Configurator::Node > & node () const
- inline::google::protobuf::RepeatedPtrField< ::Configurator::Node > ∗ mutable_node ()
- int link_size () const
- void clear_link ()
- const ::Configurator::Link & link (int index) const

- inline::Configurator::Link * mutable_link (int index)
- inline::Configurator::Link * add_link ()
- const ::google::protobuf::RepeatedPtrField< ::Configurator::Link > & link () const
- inline::google::protobuf::RepeatedPtrField< ::Configurator::Link > * mutable_link ()

7.5.2 Static Public Member Functions

- static const ::google::protobuf::Descriptor * descriptor ()
- static const Cluster & default_instance ()

7.5.3 Static Public Attributes

- static const int kNodeFieldNumber = 1
- static const int kLinkFieldNumber = 2

7.5.4 Friends

- void protobuf_AddDesc_confcluster_2eproto ()
- void protobuf_AssignDesc_confcluster_2eproto ()
- void protobuf_ShutdownFile_confcluster_2eproto ()

7.5.5 Constructor & Destructor Documentation

Configurator::Cluster::Cluster ( )

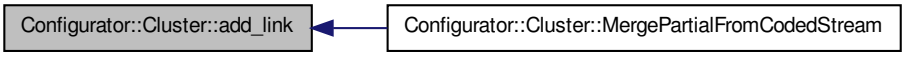
Configurator::Cluster::~Cluster ( ) `[virtual]`

Configurator::Cluster::Cluster ([const Cluster &]from)

7.5.6 Member Function Documentation

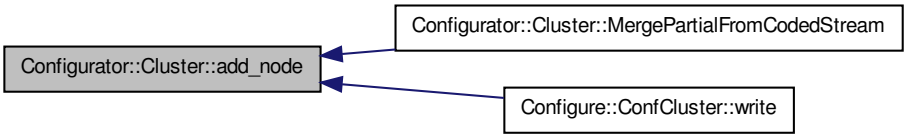
Configurator::Link * Configurator::Cluster::add_link ( ) `[inline]`

Here is the caller graph for this function:



Configurator::Node * Configurator::Cluster::add_node ( ) [inline]

Here is the caller graph for this function:



int Configurator::Cluster::ByteSize ( ) const

void Configurator::Cluster::Clear ( )

void Configurator::Cluster::clear_link ( ) [inline]

void Configurator::Cluster::clear_node ( ) [inline]

void Configurator::Cluster::CopyFrom ([const Cluster &]from)

void Configurator::Cluster::CopyFrom ([const ::google::protobuf::Message &]from)

Here is the caller graph for this function:

const Cluster & Configurator::Cluster::default_instance ( ) [static]

const ::google::protobuf::Descriptor * Configurator::Cluster::descriptor ( ) [static]

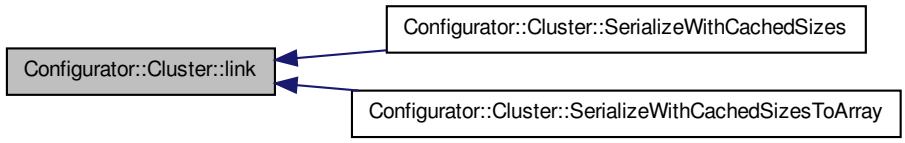
int Configurator::Cluster::GetCachedSize ( ) const [inline]

google::protobuf::Metadata Configurator::Cluster::GetMetadata ( ) const

bool Configurator::Cluster::IsInitialized ( ) const

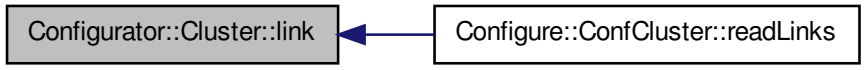
const ::google::protobuf::RepeatedPtrField<::Configurator::Link > & Configurator::Cluster::link ( ) const [inline]

Here is the caller graph for this function:



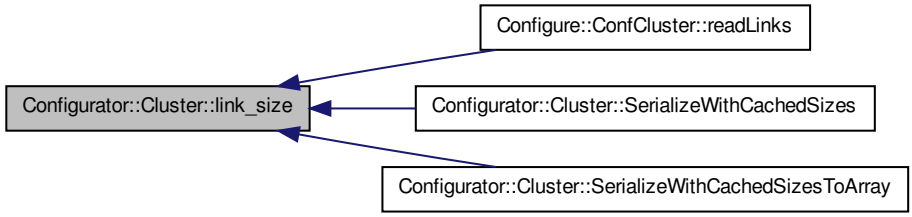
const ::Configurator::Link & Configurator::Cluster::link ([int]index) const `[inline]`

Here is the caller graph for this function:



int Configurator::Cluster::link_size ( ) const `[inline]`

Here is the caller graph for this function:

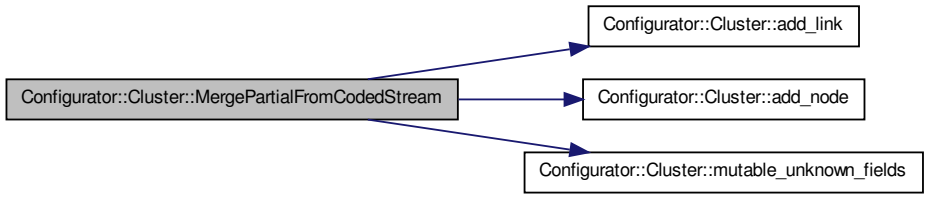


void Configurator::Cluster::MergeFrom ([const ::google::protobuf::Message &]from)

void Configurator::Cluster::MergeFrom ([const Cluster &]from)

bool Configurator::Cluster::MergePartialFromCodedStream ([::google::protobuf::io::CodedInputStream ∗]input)

Here is the call graph for this function:



google::protobuf::RepeatedPtrField<::Configurator::Link > * Configurator::Cluster::mutable_link ( ) [inline]

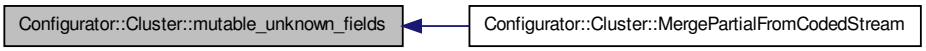
Configurator::Link * Configurator::Cluster::mutable_link ([int]index) [inline]

Configurator::Node * Configurator::Cluster::mutable_node ([int]index) [inline]

google::protobuf::RepeatedPtrField<::Configurator::Node > * Configurator::Cluster::mutable_node ( ) [inline]

inline ::google::protobuf::UnknownFieldSet* Configurator::Cluster::mutable_-unknown_fields ( ) [inline]

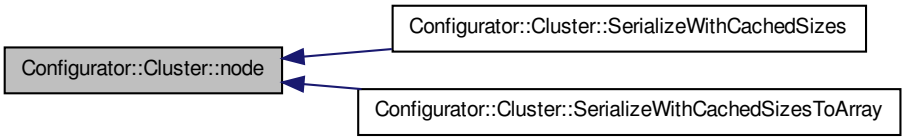
Here is the caller graph for this function:



Cluster * Configurator::Cluster::New ( ) const

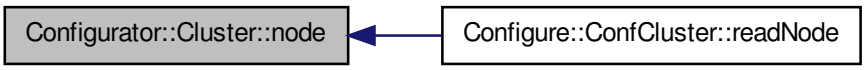
const ::google::protobuf::RepeatedPtrField<::Configurator::Node > & Configurator::Cluster::node ( ) const [inline]

Here is the caller graph for this function:



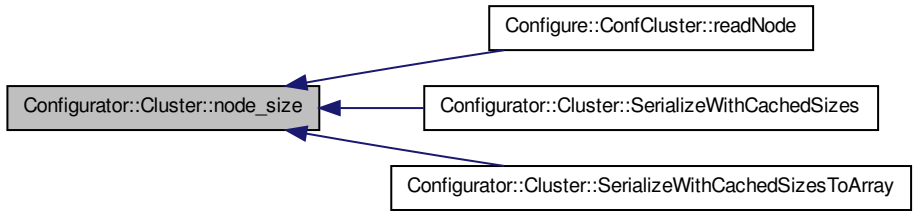
const ::Configurator::Node & Configurator::Cluster::node ([int]index) const [inline]

Here is the caller graph for this function:



int Configurator::Cluster::node_size ( ) const [inline]

Here is the caller graph for this function:

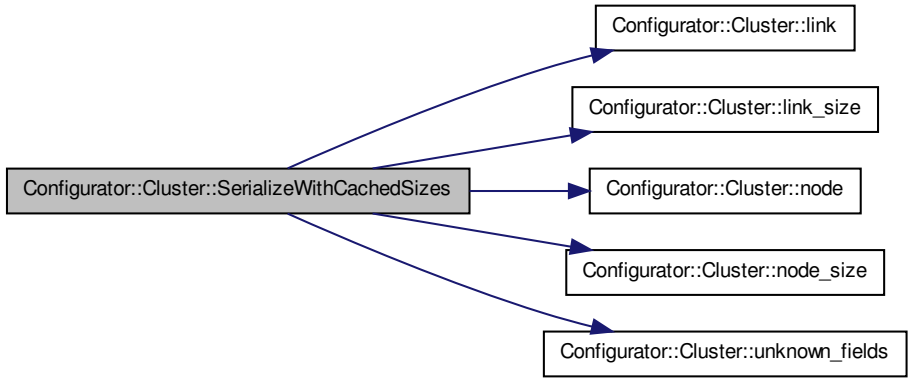


Cluster& Configurator::Cluster::operator= ([const Cluster &]from) `[inline]`

Here is the call graph for this function:

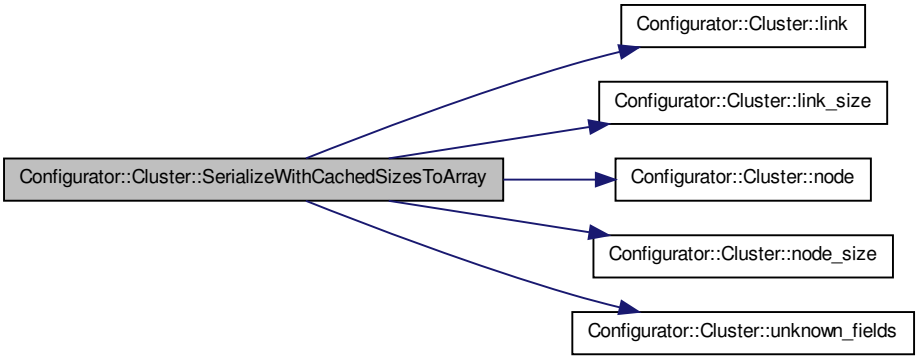
void Configurator::Cluster::SerializeWithCachedSizes ([::google::protobuf::io::CodedOutputStream *]output) const

Here is the call graph for this function:



google::protobuf::uint8 * Configurator::Cluster::SerializeWithCachedSizesToArray ([::google::protobuf::uint8 *]output) const

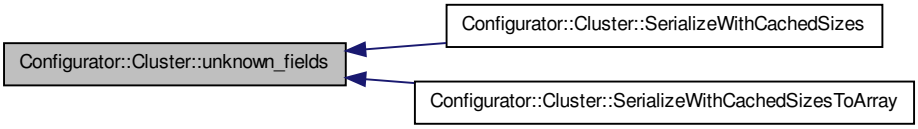
Here is the call graph for this function:



void Configurator::Cluster::Swap ([Cluster ∗]other)

const ::google::protobuf::UnknownFieldSet& Configurator::Cluster::unknown_fields ( ) const [inline]

Here is the caller graph for this function:



7.5.7 Friends And Related Function Documentation

void protobuf_AddDesc_confcluster_2eproto ( ) [friend]

void protobuf_AssignDesc_confcluster_2eproto ( ) [friend]

void protobuf_ShutdownFile_confcluster_2eproto ( ) [friend]

7.5.8 Member Data Documentation

const int Configurator::Cluster::kLinkFieldNumber = 2 [static]

const int Configurator::Cluster::kNodeFieldNumber = 1 [static]

The documentation for this class was generated from the following files:

- confcluster.pb.h
- confcluster.pb.cc

7.6 Reconfiguration::Communicator Class Reference

`#include <Communicator.h>`

7.6.1 Public Member Functions

- Communicator ()

  Constructor of the Communicator class.

- Communicator (MPI_Comm)

  Constructor of the Communicator class.

- virtual ∼Communicator ()

  virtual destructor of the class Communicator.

- int probe (int, int ∗, int ∗, MPI_Status ∗)

  Tests if there is a MPI dataframe to receive.

- int getCount (MPI_Status, MPI_Datatype, int ∗)

  Counts the number of MPI_Datatypes to be received in the next MPI_Receive() operation.

7.6.2 Public Attributes

- MPI_Comm handler

7.6.3 Detailed Description

Improves the functionality of the MPI layer by adding other functions needed for the framework.

7.6.4 Constructor & Destructor Documentation

Reconfiguration::Communicator::Communicator ( )

Constructor of the Communicator class.

Returns

∗this

Creates an object Communicator.

Reconfiguration::Communicator::Communicator ([MPI_Comm]handler)

Constructor of the Communicator class.

Parameters

in	*handler*	Handler of the MPI communication.

Returns

∗this

Creates an object Communicator and sets the handler of the communication.

Parameters

in	*handler*	Handler of the MPI communication.

Returns

∗this

Creates an object Communicator and sets the handler. of the communication.

Reconfiguration::Communicator::∼Communicator ( ) `[virtual]`

virtual destructor of the class Communicator.

Returns

void

Destroys the Communicator object.

7.6.5 Member Function Documentation

int Reconfiguration::Communicator::getCount ([MPI_Status]status, MPI_Datatype *type,* int * *iSize*)

Counts the number of MPI_Datatypes to be received in the next MPI_Receive() operation.

Parameters

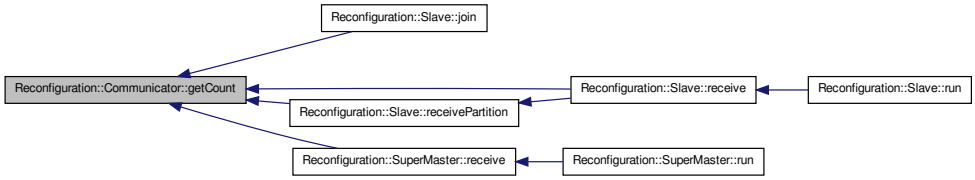
in	*status*	Status variable of the last probe operation.
in	*type*	Handler of the MPI communication.
out	*iSize*	Number of MPI_Datatype type elements.

Returns

*this

If we have to allocate memory for receiving a dataframe of an unknown size, we have to know first the number of elements of an specific MPI_Datatype that we are going to receive. This function calculates the number of this MPI_Datatype elements to being able to allocate the memory to receive this dataframe.

Here is the caller graph for this function:



int Reconfiguration::Communicator::probe ([int]iSource, int * *iTag,* int * *src,* MPI_Status * *status*)

Tests if there is a MPI dataframe to receive.

Parameters

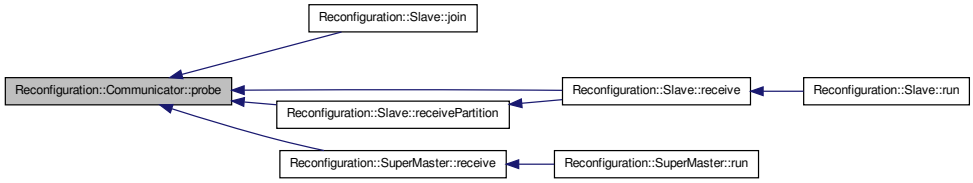
in	*iSource*	Source of the dataframes to receive.
out	*iTag*	Tag of the dataframe received.
out	*src*	Identifier of the source process (it only has sens if the first parameter is MPI_ANY_SOURCE).
out	*status*	Status of the MPI operation.

Returns

*this

Checks whether a certain process has something to receive from an specific process (source). If there is nothing to receive, the function waits until a new dataframe is available.

Here is the caller graph for this function:



7.6.6 Member Data Documentation

MPI_Comm Reconfiguration::Communicator::handler

Handler of the communication.

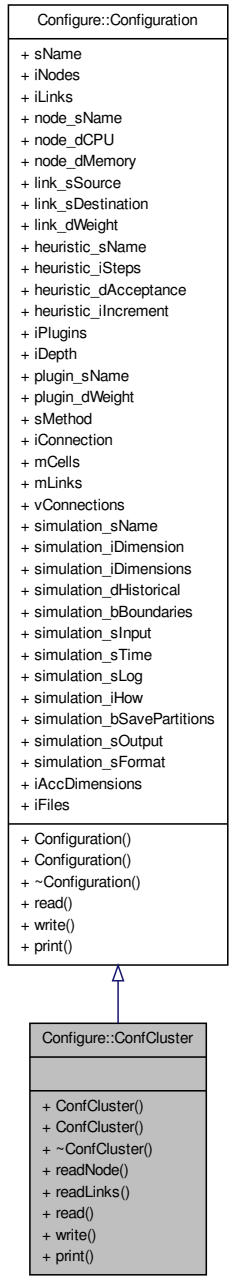
The documentation for this class was generated from the following files:

- Communicator.h
- Communicator.cpp

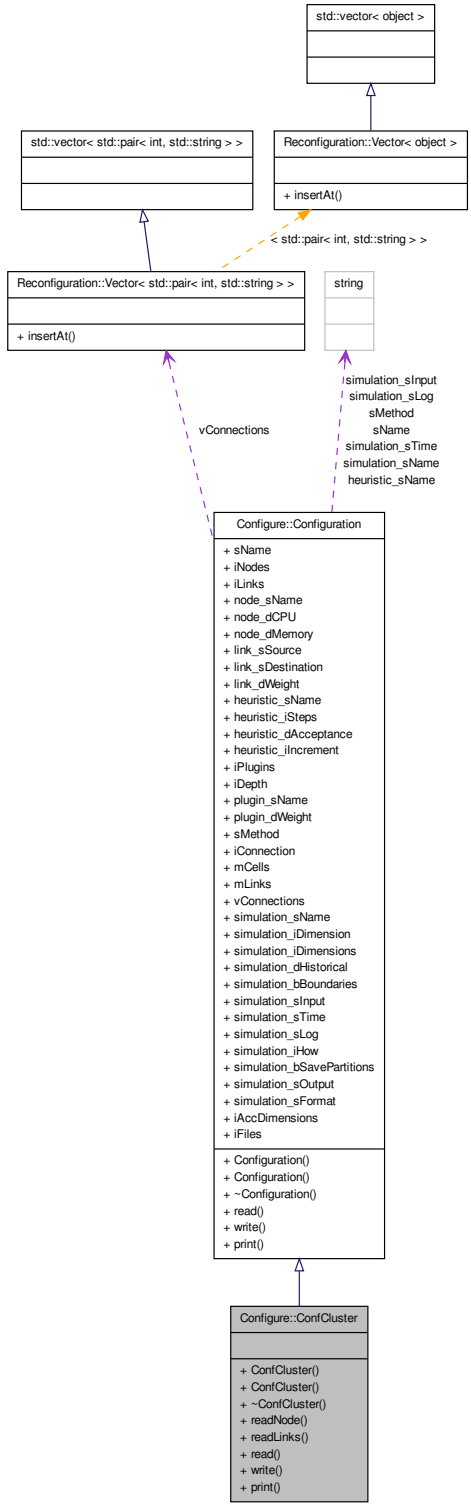
7.7 Configure::ConfCluster Class Reference

`#include <Configuration.h>`

Inheritance diagram for Configure::ConfCluster:



Collaboration diagram for Configure::ConfCluster:



7.7.1 Public Member Functions

- ConfCluster ()

  Constructor of the ConfCluster class.

- ConfCluster (std::string)

  Constructor of the ConfCluster class.

- virtual ~ConfCluster ()

  virtual destructor of the class ConfCluster.

- void readNode (Configurator::Cluster)

  Reads the cluster nodes structure of the configuration file.

- void readLinks (Configurator::Cluster)

  Reads the links structure between nodes of the configuration file.

- void read ()

  Reads the configuration file for the cluster specifications.

- void write ()

  Writes the configuration file for the cluster specifications.

- void print ()

  Prints the ConfCluster object.

7.7.2 Detailed Description

Class for managing the cluster configuration file (tipically cluster.conf file).

7.7.3 Constructor & Destructor Documentation

Configure::ConfCluster::ConfCluster ( )

Constructor of the ConfCluster class.

Returns

∗this

Creates an object ConfCluster.

Configure::ConfCluster::ConfCluster ([std::string]sName)

Constructor of the ConfCluster class.

Parameters

in	*sName*	Name of the simulation.

Returns

∗this

Creates an object ConfCluster and sets the name chosen by the user.

Configure::ConfCluster::∼ConfCluster ( ) `[virtual]`

virtual destructor of the class ConfCluster.

Returns

void

Destroys the ConfCluster object and shuts down the protobuffers library.

7.7.4 Member Function Documentation

void Configure::ConfCluster::print ( ) `[virtual]`

Prints the ConfCluster object.

Returns

void

Prints the properties of the ConfCluster object.

Implements Configure::Configuration.

void Configure::ConfCluster::read ( ) [virtual]

Reads the configuration file for the cluster specifications.

Returns

void

Reads the configuration file for the cluster specifications.

Implements Configure::Configuration.

void Configure::ConfCluster::readLinks ([Configurator::Cluster]cluster)

Reads the links structure between nodes of the configuration file.

Parameters

in	*cluster*	Cluster structure.

Returns

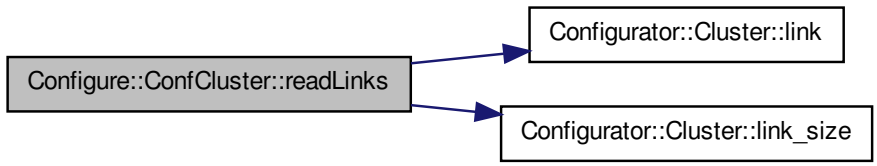
void

Reads the links parameters from the configuration file.

Exceptions

ConfigurationException	Error while reading the configuration file: values not recognized, invalid arguments...

Here is the call graph for this function:



void Configure::ConfCluster::readNode ([Configurator::Cluster]cluster)

Reads the cluster nodes structure of the configuration file.

Parameters

in	*cluster*	Cluster structure.

Returns

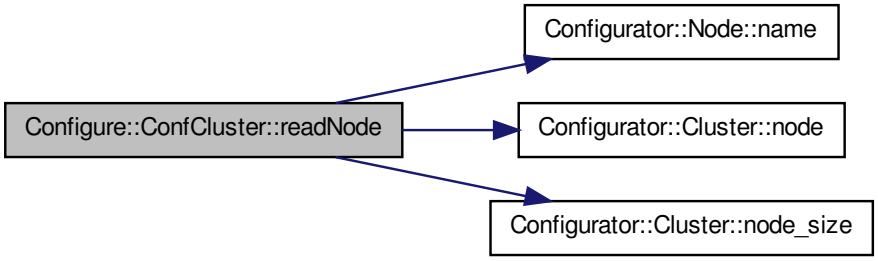
void

Reads the nodes parameters from the configuration file.

Exceptions

ConfigurationException	Error while reading the configuration `file:` values not recognized, invalid arguments...

Here is the call graph for this function:



void Configure::ConfCluster::write ( ) `[virtual]`

Writes the configuration file for the cluster specifications.

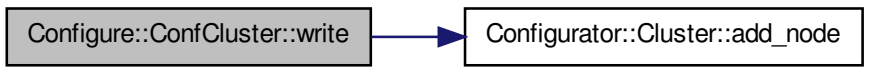
Returns

void

Writes the configuration file for the cluster specifications. The cluster configuration file can be directly written. This is only an example of how to write a configuration file using the platform.

Implements Configure::Configuration.

Here is the call graph for this function:



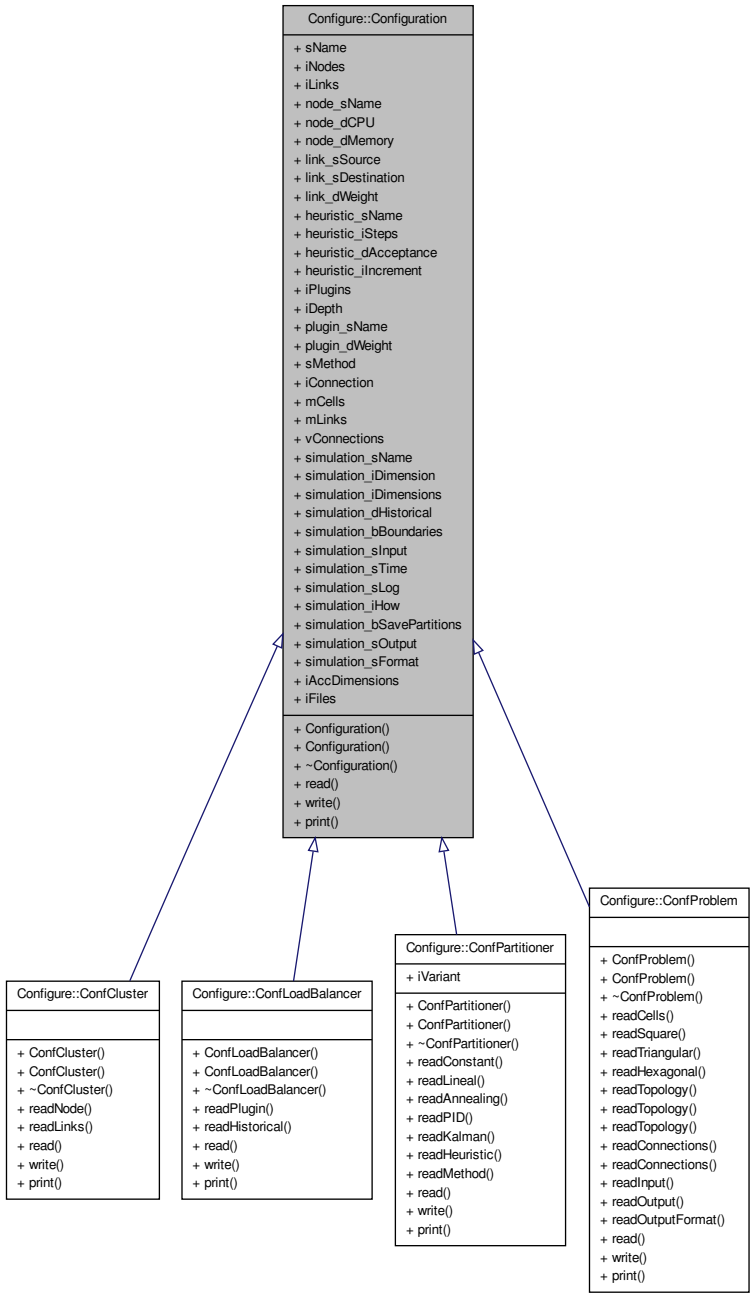
The documentation for this class was generated from the following files:

- Configuration.h
- Configuration.cpp

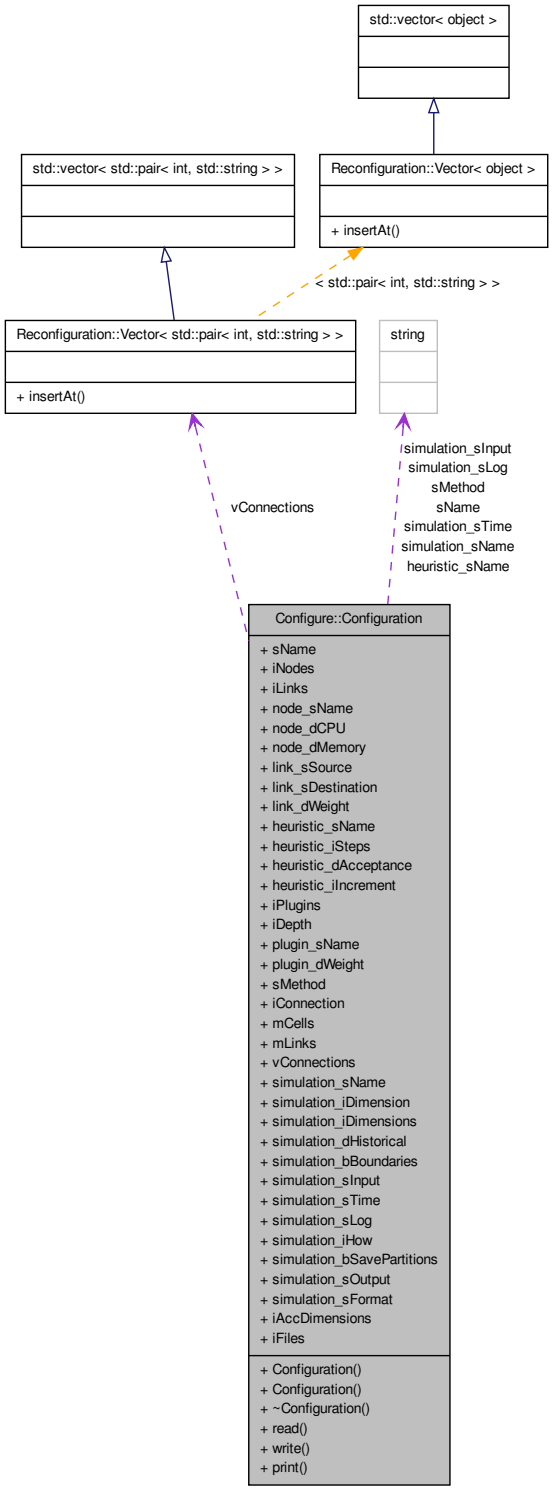
7.8 Configure::Configuration Class Reference

`#include <Configuration.h>`

Inheritance diagram for Configure::Configuration:



Collaboration diagram for Configure::Configuration:



7.8.1 Public Member Functions

- Configuration ()

  Constructor of the Configuration class.

- Configuration (std::string)

  Constructor of the Configuration class.

- virtual ~Configuration ()

  virtual destructor of the class Configuration.

- virtual void read ()=0

  Abstract reader of a configuration file.

- virtual void write ()=0

  Abstract writer of a configuration file.

- virtual void print ()=0

  Abstract displayer of a configuration file.

7.8.2 Public Attributes

- std::string sName
- unsigned int iNodes
- unsigned int iLinks
- std::string * node_sName
- double * node_dCPU
- double * node_dMemory
- std::string * link_sSource
- std::string * link_sDestination
- double * link_dWeight
- std::string heuristic_sName
- int heuristic_iSteps
- double heuristic_dAcceptance
- int heuristic_iIncrement

- unsigned int iPlugins
- unsigned int iDepth
- std::string * plugin_sName
- double * plugin_dWeight
- std::string sMethod
- unsigned int iConnection
- std::map< std::string, Reconfiguration::Coordinate > mCells
- std::map< std::string, int > mLinks
- Vector< std::pair< int, std::string > > vConnections
- std::string simulation_sName
- int simulation_iDimension
- int * simulation_iDimensions
- double * simulation_dHistorical
- bool * simulation_bBoundaries
- std::string simulation_sInput
- std::string simulation_sTime
- std::string simulation_sLog
- int simulation_iHow
- bool simulation_bSavePartitions
- std::string * simulation_sOutput
- std::string * simulation_sFormat
- int * iAccDimensions
- int iFiles

7.8.3 Constructor & Destructor Documentation

Configure::Configuration::Configuration ( )

Constructor of the Configuration class.

Returns

*this

Creates an object Configuration.

Configure::Configuration::Configuration ([std::string]sName)

Constructor of the Configuration class.

Parameters

in	*sName*	Name of the simulation.

Returns

∗this

Creates an object Configuration and sets the name chosen by the user. Verifies that the version of the protobuffer library we linked against is compatible with the version of the headers we compiled against.

Configure::Configuration::∼Configuration ( ) [virtual]

virtual destructor of the class Configuration.

Returns

void

Destroys the Configuration object and shuts down the protobuffers library.

7.8.4 Member Function Documentation

virtual void Configure::Configuration::print ( ) [pure virtual]

Abstract displayer of a configuration file.

Returns

void

Abstract displayer of a configuration file. Implemented at all possible configuration classes.

Implemented in Configure::ConfCluster, Configure::ConfLoadBalancer, Configure::ConfPartitioner and Configure::ConfProblem.

virtual void Configure::Configuration::read ( ) [pure virtual]

Abstract reader of a configuration file.

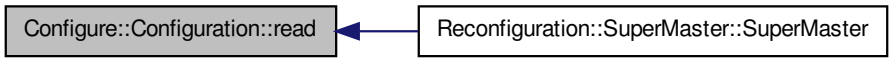
Returns

void

Abstract reader of a configuration file. Implemented at all possible configuration classes.

Implemented in Configure::ConfCluster, Configure::ConfLoadBalancer, Configure::ConfPartitioner, and Configure::ConfProblem.

Here is the caller graph for this function:



virtual void Configure::Configuration::write ( ) [pure virtual]

Abstract writer of a configuration file.

Returns

void

Abstract writer of a configuration file. Implemented at all possible configuration classes.

Implemented in Configure::ConfCluster, Configure::ConfLoadBalancer, Configure::ConfPartitioner, and Configure::ConfProblem.

7.8.5 Member Data Documentation

double Configure::Configuration::heuristic_dAcceptance

Acceptance threshold for accepting a certain domain decomposition.

int Configure::Configuration::heuristic_iIncrement

For heuristic methods based on annealings, the temperature increment.

int Configure::Configuration::heuristic_iSteps

Number of steps between two consecutives calls to the partition method.

std::string Configure::Configuration::heuristic_sName

Name of the heuristic method.

int * Configure::Configuration::iAccDimensions

Array of accumulated length per problem dimension. The last position of this array must be equal to the number of vertices of the graph.

unsigned int Configure::Configuration::iConnection

Number of connections defined in the problem configuration file.

unsigned int Configure::Configuration::iDepth

Number of historical values to take into account to calculate the loadbalancing policies, if any time plugin is involved.

int Configure::Configuration::iFiles

Number of output files.

unsigned int Configure::Configuration::iLinks

Number of links of the system graph.

unsigned int Configure::Configuration::iNodes

Number of vertices of the system graph.

unsigned int Configure::Configuration::iPlugins

Number of loadbalacing plugins.

double * Configure::Configuration::link_dWeight

Weight of the link.

std::string * Configure::Configuration::link_sDestination

Hostname of the destination of the link.

std::string * Configure::Configuration::link_sSource

Hostname of the source of the link.

std::map< std::string, Reconfiguration::Coordinate > Configure::Configuration::mCells

Pairs between link name and set of coordenates for that link.

std::map< std::string, int > Configure::Configuration::mLinks

Pairs between links names and global offset for that link. This offset depends on the graph dimensions.

double * Configure::Configuration::node_dCPU

Nodes frequency.

double * Configure::Configuration::node_dMemory

Nodes available memory.

std::string * Configure::Configuration::node_sName

Nodes hostname.

double * Configure::Configuration::plugin_dWeight

Weight of each loaded plugin.

std::string * Configure::Configuration::plugin_sName

Name of the plugins to be loaded.

bool * Configure::Configuration::simulation_bBoundaries

Boundary policy per problems dimension.

bool Configure::Configuration::simulation_bSavePartitions

Whether the partition shapes must be save into an image.

double * Configure::Configuration::simulation_dHistorical

Weight of each historical value from the point of view of the global weight for the temporal plugin. Related to the values read from the user configuration file.

int Configure::Configuration::simulation_iDimension

Number of dimensions of the problem.

int * Configure::Configuration::simulation_iDimensions

Length per dimension.

int Configure::Configuration::simulation_iHow

Save all results in one file or one slave per file.

See also

WHICH

std::string * Configure::Configuration::simulation_sFormat

Format extensions for the output files.

std::string Configure::Configuration::simulation_sInput

Filename for input data.

std::string Configure::Configuration::simulation_sLog

Filename for activity log.

std::string Configure::Configuration::simulation_sName

Name of the simulation.

std::string * Configure::Configuration::simulation_sOutput

Names for the output files. More than one file can be generated per step.

std::string Configure::Configuration::simulation_sTime

Filename for time log.

std::string Configure::Configuration::sMethod

Domain decomposition method name.

std::string Configure::Configuration::sName

Name of the execution (binary file, output files...)

Vector< std::pair< int, std::string > > Configure::Configuration::vConnections

Pairs between vertex identifiers and link names.

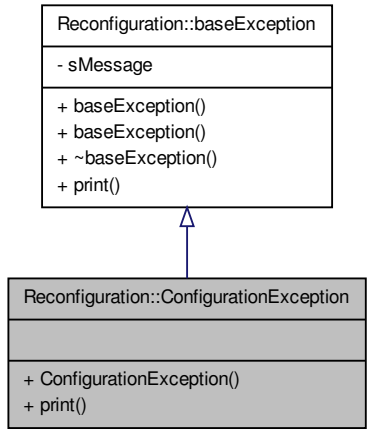
The documentation for this class was generated from the following files:

- Configuration.h
- Configuration.cpp

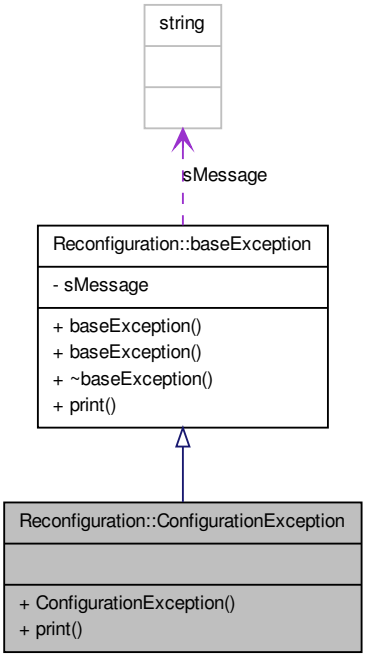
7.9 Reconfiguration::ConfigurationException Class Reference

`#include <Exception.h>`

Inheritance diagram for Reconfiguration::ConfigurationException:



Collaboration diagram for Reconfiguration::ConfigurationException:



7.9.1 Public Member Functions

- ConfigurationException (std::string)

  Constructor of the ConfigurationException class.

- void print ()

  Prints the ConfigurationException exception.

7.9.2 Detailed Description

Exception thrown if a configuration error occurs.

7.9.3 Constructor & Destructor Documentation

Reconfiguration::ConfigurationException::ConfigurationException ([std::string]sText)

Constructor of the ConfigurationException class.

Parameters

in	*sText*	Configuration error message.

Returns

∗this

Creates an object ConfigurationException calling the baseException constructor and sets the message of the exception.

Parameters

in	*sText*	Configuration error text.

Returns

∗this

Creates an object ConfigurationException calling the baseException constructor and sets the message of the exception.

7.9.4 Member Function Documentation

void Reconfiguration::ConfigurationException::print ( )

Prints the ConfigurationException exception.

Returns

void

Prints the exception by printing its message.

Reimplemented from Reconfiguration::baseException.

The documentation for this class was generated from the following files:

- Exception.h
- Exception.cpp

7.10 Configure Class Reference

`#include <Configuration.h>`

7.10.1 Detailed Description

Base configuration, father of all the configuration files that the user need to write before launching the platform.

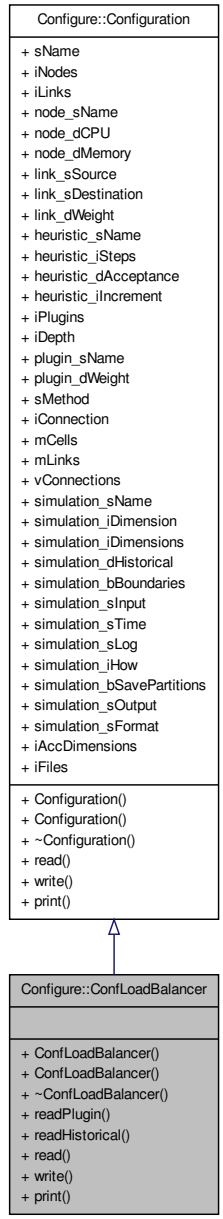
The documentation for this class was generated from the following file:

- Configuration.h

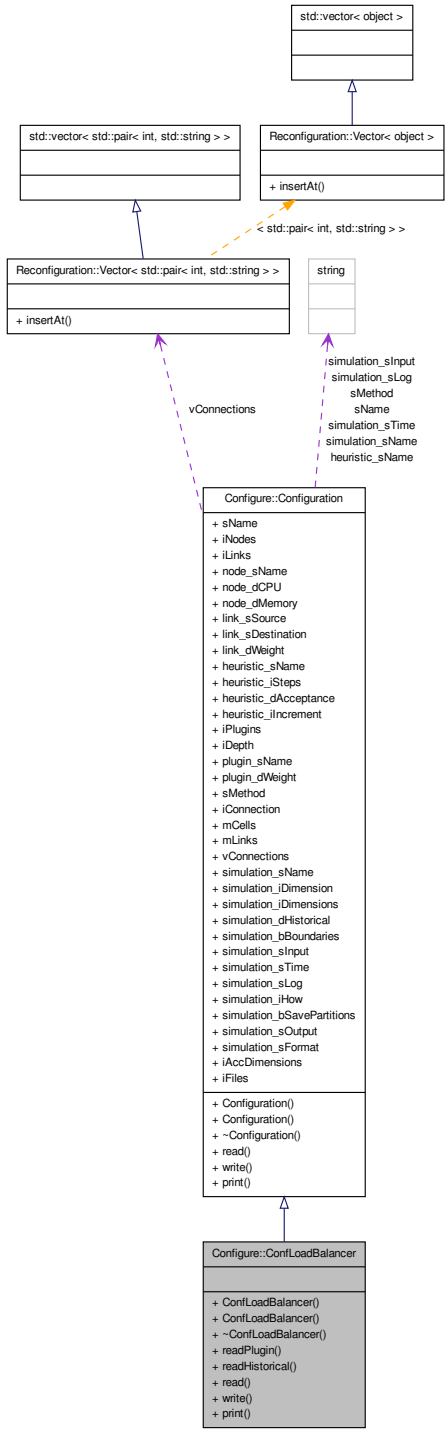
7.11 Configure::ConfLoadBalancer Class Reference

`#include <Configuration.h>`

Inheritance diagram for Configure::ConfLoadBalancer:



Collaboration diagram for Configure::ConfLoadBalancer:



7.11.1 Public Member Functions

- ConfLoadBalancer ()

  Constructor of the ConfLoadBalancer class.

- ConfLoadBalancer (std::string)

  Constructor of the ConfLoadBalancer class.

- virtual ~ConfLoadBalancer ()

  virtual destructor of the class ConfLoadBalancer.

- void readPlugin (Configurator::LoadBalancer)

  Reads the plugin structures of the configuration file.

- void readHistorical (Configurator::LoadBalancer)

  Reads the historical structure of the configuration file if it was defined.

- void read ()

  Reads the configuration file for the loadbalancer specifications.

- void write ()

  Writes the configuration file for the loadbalancer specifications.

- void print ()

  Prints the ConfLoadBalancer object.

7.11.2 Detailed Description

Class for managing the loadbalancer configuration file (tipically loadbalancer.conf file).

7.11.3 Constructor & Destructor Documentation

Configure::ConfLoadBalancer::ConfLoadBalancer ( )

Constructor of the ConfLoadBalancer class.

Returns

∗this

Creates an object ConfLoadBalancer.

Configure::ConfLoadBalancer::ConfLoadBalancer ([std::string]sName)

Constructor of the ConfLoadBalancer class.

Parameters

in	*sName*	Name of the simulation.

Returns

∗this

Creates an object ConfLoadBalancer and sets the name chosen by the user.

Configure::ConfLoadBalancer::∼ConfLoadBalancer ( ) [virtual]

virtual destructor of the class ConfLoadBalancer.

Returns

void

Destroys the ConfLoadBalancer object and shuts down the protobuffers library.

7.11.4 Member Function Documentation

void Configure::ConfLoadBalancer::print ( ) [virtual]

Prints the ConfLoadBalancer object.

Returns

void

Prints the properties of the ConfLoadBalancer object.

Implements Configure::Configuration.

void Configure::ConfLoadBalancer::read ( ) [virtual]

Reads the configuration file for the loadbalancer specifications.

Returns

void

Reads the configuration file for the loadbalancer specifications.

Implements Configure::Configuration.

void Configure::ConfLoadBalancer::readHistorical ([Configurator::LoadBalancer]loadbalancer)

Reads the historical structure of the configuration file if it was defined.

Parameters

in	*loadbalancer*	Loadbalancer structure.

Returns

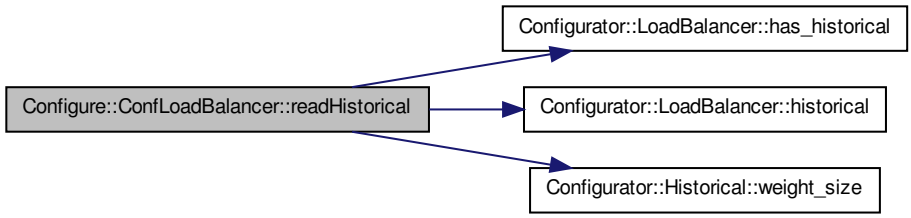
void

Reads the historical parameter values from the configuration file.

Exceptions

ConfigurationException	Error while reading the configuration file: values not recognized, invalid arguments...

Here is the call graph for this function:



void Configure::ConfLoadBalancer::readPlugin ([Configurator::LoadBalancer]loadbalancer)

Reads the plugin structures of the configuration file.

Parameters

in	*loadbalancer*	Loadbalancer structure.

Returns

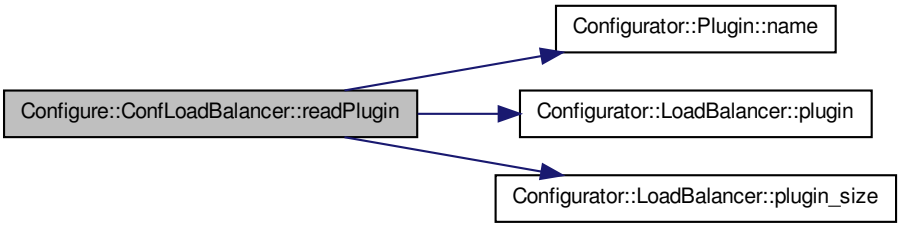
void

Reads the plugins parameters from the configuration file.

Exceptions

ConfigurationException	Error while reading the configuration `file:` values not recognized, invalid arguments...

Here is the call graph for this function:



void Configure::ConfLoadBalancer::write ( ) `[virtual]`

Writes the configuration file for the loadbalancer specifications.

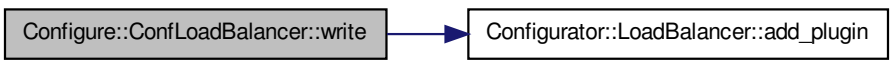
Returns

void

Writes the configuration file for the loadbalancer specifications. The cluster configuration file can be directly written. This is only an example of how to write a configuration file using the platform.

Implements Configure::Configuration.

Here is the call graph for this function:



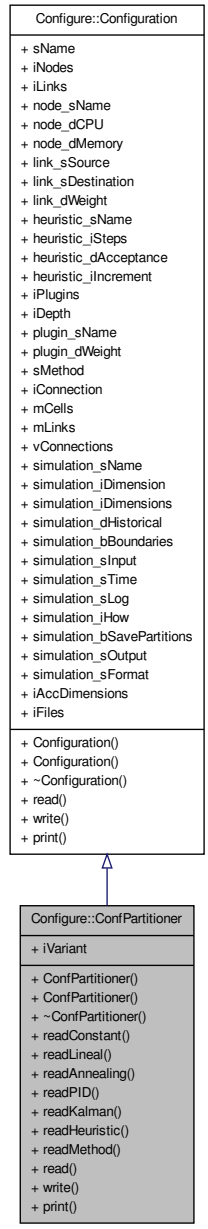
The documentation for this class was generated from the following files:

- Configuration.h
- Configuration.cpp

7.12 Configure::ConfPartitioner Class Reference

`#include <Configuration.h>`

Inheritance diagram for Configure::ConfPartitioner:



Collaboration diagram for Configure::ConfPartitioner:

7.12.1 Public Member Functions

- ConfPartitioner ()

  Constructor of the ConfPartitioner class.

- ConfPartitioner (std::string)

  Constructor of the ConfPartitioner class.

- virtual ~ConfPartitioner ()

  virtual destructor of the class ConfPartitioner.

- void readConstant (Configurator::Heuristic)

  Reads the heuristic constant structure.

- void readLineal (Configurator::Heuristic)

  Reads the heuristic lineal structure.

- void readAnnealing (Configurator::Heuristic)

  Reads the heuristic annealing structure.

- void readPID (Configurator::Heuristic)

  Reads the heuristic PID structure.

- void readKalman (Configurator::Heuristic)

  Reads the heuristic Kalman structure.

- void readHeuristic (Configurator::Partitioner)

  Reads the heuristic structure.

- void readMethod (Configurator::Partitioner)

  Reads the partitioner method structure.

- void read ()

  Reads the configuration file for the partitioner specifications.

- void write ()

  Writes the configuration file for the partitioner specifications.

- void print ()

  Prints the ConfPartitioner object.

7.12.2 Public Attributes

- int iVariant

7.12.3 Detailed Description

Class for managing the domain decomposition configuration file (tipically partitioner.conf file).

7.12.4 Constructor & Destructor Documentation

Configure::ConfPartitioner::ConfPartitioner ( )

Constructor of the ConfPartitioner class.

Returns

∗this

Creates an object ConfPartitioner.

Configure::ConfPartitioner::ConfPartitioner ([std::string]sName)

Constructor of the ConfPartitioner class.

Parameters

in	*sName*	Name of the simulation.

Returns

∗this

Creates an object ConfPartitioner and sets the name chosen by the user.

Configure::ConfPartitioner::∼ConfPartitioner ( ) `[virtual]`

virtual destructor of the class ConfPartitioner.

Returns

void

Destroys the ConfPartitioner object and shuts down the protobuffers library.

7.12.5 Member Function Documentation

void Configure::ConfPartitioner::print ( ) [virtual]

Prints the ConfPartitioner object.

Returns

void

Prints the properties of the ConfPartitioner object.

Implements Configure::Configuration.

void Configure::ConfPartitioner::read ( ) [virtual]

Reads the configuration file for the partitioner specifications.

Returns

void

Reads the configuration file for the partitioner specifications.

Implements Configure::Configuration.

void Configure::ConfPartitioner::readAnnealing ([Configurator::Heuristic]heuristic)

Reads the heuristic annealing structure.

Parameters

in	*heuristic*	Heuristic structure.

Returns

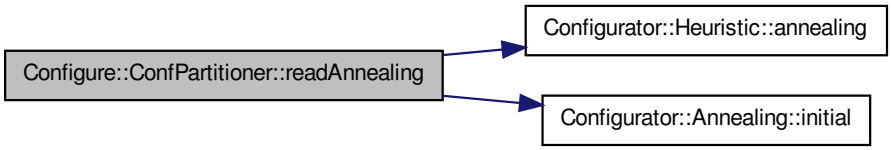
void

Reads the heuristic annealing structure from the configuration file.

Exceptions

ConfigurationException	Error while reading the configuration `file:` values not recognized, invalid arguments...

Here is the call graph for this function:



void Configure::ConfPartitioner::readConstant ([Configurator::Heuristic]heuristic)

Reads the heuristic constant structure.

Parameters

`in`	*heuristic*	Heuristic structure.

Returns

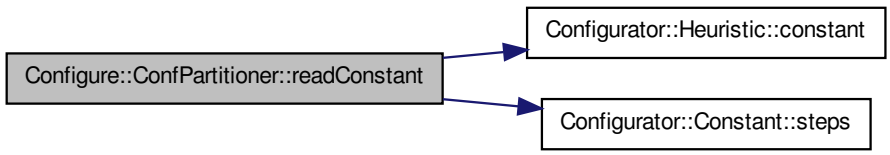
void

Reads the heuristic constant structure from the configuration file.

Exceptions

ConfigurationException	Error while reading the configuration `file:` values not recognized, invalid arguments...

Here is the call graph for this function:



void Configure::ConfPartitioner::readHeuristic ([Configurator::Partitioner]partitioner)

Reads the heuristic structure.

Parameters

`in`	*partitioner*	Partitioner structure.

Returns

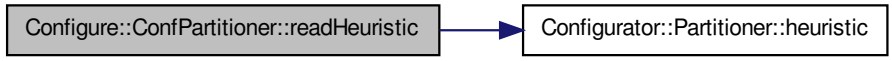
void

Reads the heuristic structure from the configuration file.

Exceptions

ConfigurationException	Error while reading the configuration `file:` values not recognized, invalid arguments...

Here is the call graph for this function:



void Configure::ConfPartitioner::readKalman ([Configurator::Heuristic]heuristic)

Reads the heuristic Kalman structure.

Parameters

`in`	*heuristic*	Heuristic structure.

Returns

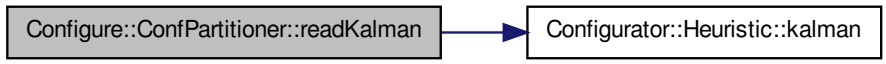
void

Reads the heuristic Kalman structure from the configuration file.

Exceptions

ConfigurationException	Error while reading the configuration `file:` values not recognized, invalid arguments...

Here is the call graph for this function:



void Configure::ConfPartitioner::readLineal ([Configurator::Heuristic]heuristic)

Reads the heuristic lineal structure.

Parameters

`in`	*heuristic*	Heuristic structure.

Returns

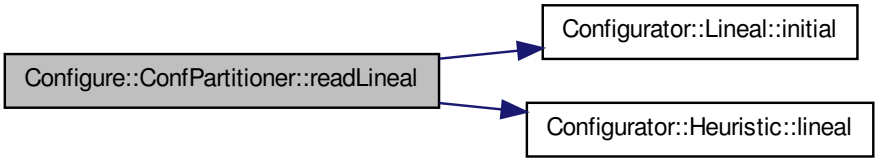
void

Reads the heuristic lineal structure from the configuration file.

Exceptions

ConfigurationException	Error while reading the configuration `file:` values not recognized, invalid arguments...

Here is the call graph for this function:



void Configure::ConfPartitioner::readMethod ([Configurator::Partitioner]partitioner)

Reads the partitioner method structure.

Parameters

`in`	*partitioner*	Partitioner structure.

Returns

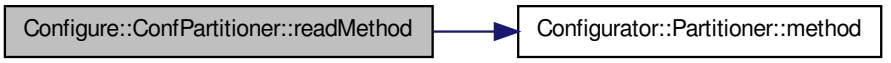
void

Reads the partitioner method structure from the configuration file.

Exceptions

ConfigurationException	Error while reading the configuration `file:` values not recognized, invalid arguments...

Here is the call graph for this function:



void Configure::ConfPartitioner::readPID ([Configurator::Heuristic]heuristic)

Reads the heuristic PID structure.

Parameters

`in`	*heuristic*	Heuristic structure.

Returns

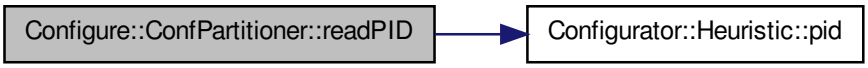
void

Reads the heuristic PID structure from the configuration file.

Exceptions

ConfigurationException	Error while reading the configuration `file:` values not recognized, invalid arguments...

Here is the call graph for this function:



void Configure::ConfPartitioner::write ( ) `[virtual]`

Writes the configuration file for the partitioner specifications.

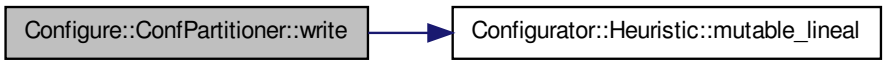
Returns

void

Writes the configuration file for the partitioner specifications. The cluster configuration file can be directly written. This is only an example of how to write a configuration file using the platform.

Implements Configure::Configuration.

Here is the call graph for this function:



7.12.6 Member Data Documentation

int Configure::ConfPartitioner::iVariant

Whether changes on the partition shape are allowed during the execution.

See also

VARIANTS

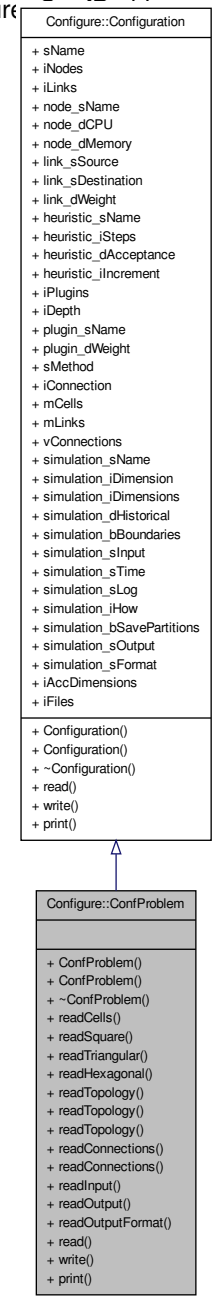
The documentation for this class was generated from the following files:

- Configuration.h
- Configuration.cpp

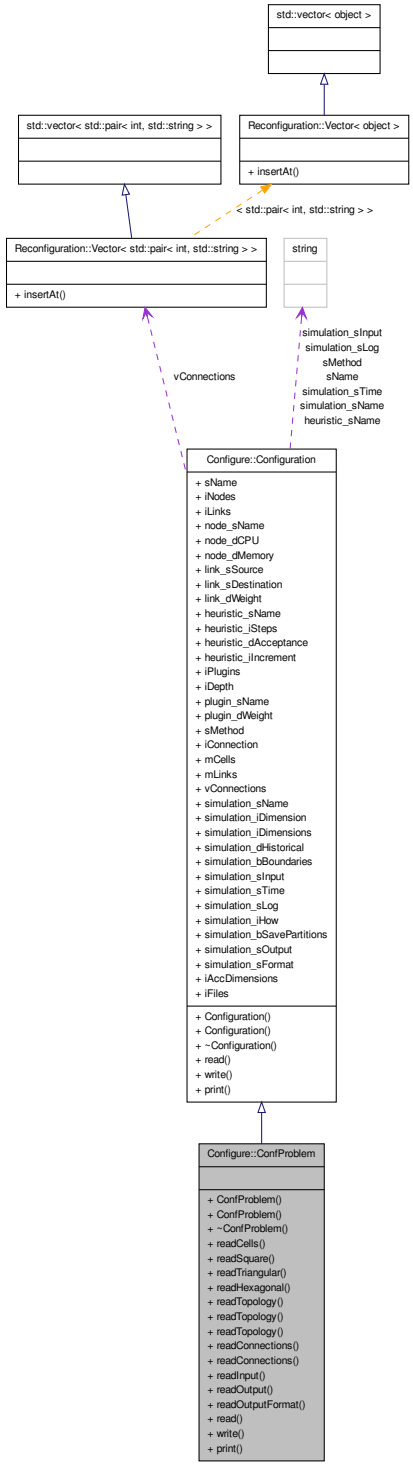
7.13 Configure::ConfProblem Class Reference

`#include <Configuration.h>`

Inheritance diagram for Configure



Collaboration diagram for Configure::ConfProblem:



7.13.1 Public Member Functions

- ConfProblem ()

  Constructor of the ConfProblem class.

- ConfProblem (std::string)

  Constructor of the ConfProblem class.

- virtual ~ConfProblem ()

  virtual destructor of the class ConfProblem.

- void readCells (Configurator::Simulation)

  Reads the cell structures.

- void readSquare (Configurator::Cell)

  Reads the Square structure.

- void readTriangular (Configurator::Cell)

  Reads the Triangular structure.

- void readHexagonal (Configurator::Cell)

  Reads the Hexagonal structure.

- void readTopology (Configurator::Simulation)

  Reads the Topology structure.

- void readTopology (Configurator::Matrix)

  Reads the Topology for a matrix structure.

- void readTopology (Configurator::Graph)

  Reads the Topology for a graph structure.

- void readConnections (Configurator::Matrix)

  Reads the connections of a matrix topology.

- void readConnections (Configurator::Graph)

  Reads the connections of a graph topology.

- void readInput (Configurator::Simulation)

  Reads the input filename for the initial problem data values.

- void readOutput (Configurator::Simulation)

  Reads the output structure.

- void readOutputFormat (Configurator::Output)

  Reads the output format structures.

- void read ()

  Reads the configuration file for the problem specifications.

- void write ()

  Writes the configuration file for the problem specifications.

- void print ()

  Prints the ConfProblem object.

7.13.2 Detailed Description

Class for managing the problem configuration file (tipically named as the problem.conf file).

7.13.3 Constructor & Destructor Documentation

Configure::ConfProblem::ConfProblem ( )

Constructor of the ConfProblem class.

Returns

∗this

Creates an object ConfProblem.

Configure::ConfProblem::ConfProblem ([std::string]sName)

Constructor of the ConfProblem class.

Parameters

in	*sName*	Name of the simulation.

Returns

∗this

Creates an object ConfProblem and sets the name chosen by the user.

Configure::ConfProblem::∼ConfProblem ( ) [virtual]

virtual destructor of the class ConfProblem.

Returns

void

Destroys the ConfProblem object and shuts down the protobuffers library.

7.13.4 Member Function Documentation

void Configure::ConfProblem::print ( ) [virtual]

Prints the ConfProblem object.

Returns

void

Prints the properties of the ConfProblem object.

Implements Configure::Configuration.

void Configure::ConfProblem::read ( ) [virtual]

Reads the configuration file for the problem specifications.

Returns

void

Reads the configuration file for the problem specifications.

Implements Configure::Configuration.

Here is the call graph for this function:

void Configure::ConfProblem::readCells ([Configurator::Simulation]simulation)

Reads the cell structures.

Parameters

in	*simulation*	Simulation structure.

Returns

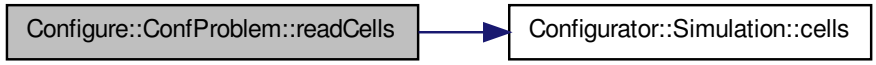
void

Reads the cell structures from the configuration file.

Exceptions

ConfigurationException	Error while reading the configuration file: values not recognized, invalid arguments...

Here is the call graph for this function:



void Configure::ConfProblem::readConnections ([Configurator::Matrix]matrix)

Reads the connections of a matrix topology.

Parameters

in	*matrix*	Matrix problem structure.

Returns

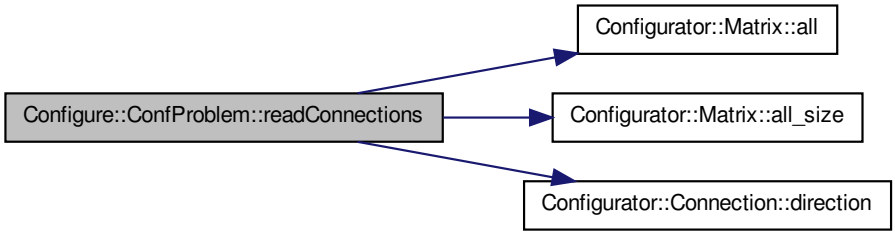
void

Reads the connections the matrix topology from the configuration file.

Exceptions

ConfigurationException	Error while reading the configuration `file:` values not recognized, invalid arguments...

Here is the call graph for this function:



void Configure::ConfProblem::readConnections ([Configurator::Graph]graph)

Reads the connections of a graph topology.

Parameters

`in`	*graph*	Graph problem structure.

Returns

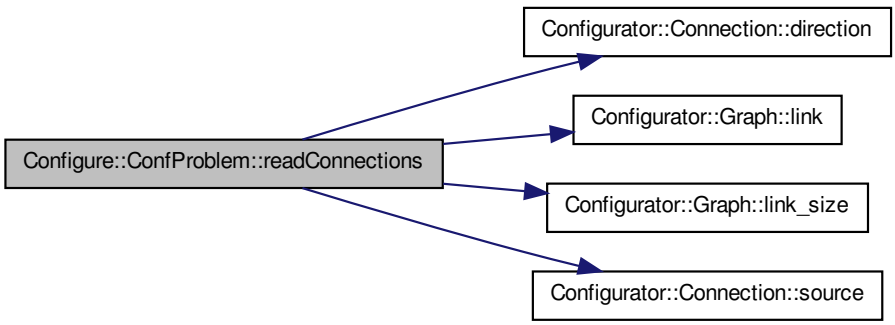
void

Reads the connections the graph topology from the configuration file.

Exceptions

ConfigurationException	Error while reading the configuration `file:` values not recognized, invalid arguments...

Here is the call graph for this function:



void Configure::ConfProblem::readHexagonal ([Configurator::Cell]cell)

Reads the Hexagonal structure.

Parameters

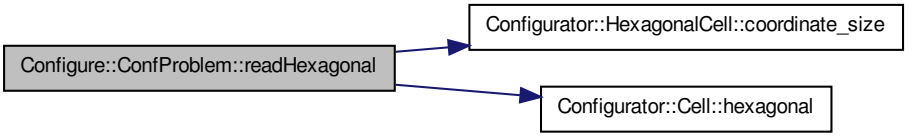
in	*cell*	Cell structure.

Returns

void

Reads the parameters of the hexagonal structure from the configuration file.

Here is the call graph for this function:



void Configure::ConfProblem::readInput ([Configurator::Simulation]simulation)

Reads the input filename for the initial problem data values.

Parameters

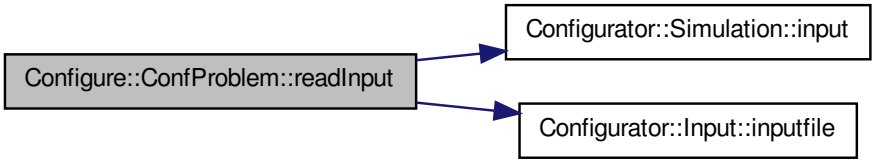
in	*simulation*	Simulation structure.

Returns

void

Reads the input filename where the initial values for the problem are stored.

Here is the call graph for this function:



void Configure::ConfProblem::readOutput ([Configurator::Simulation]simulation)

Reads the output structure.

Parameters

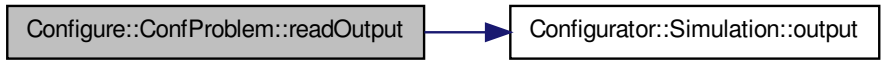
in	*simulation*	Simulation structure.

Returns

void

Reads the parameters related to the way the output is going to be made.

Here is the call graph for this function:



void Configure::ConfProblem::readOutputFormat ([Configurator::Output]output)

Reads the output format structures.

Parameters

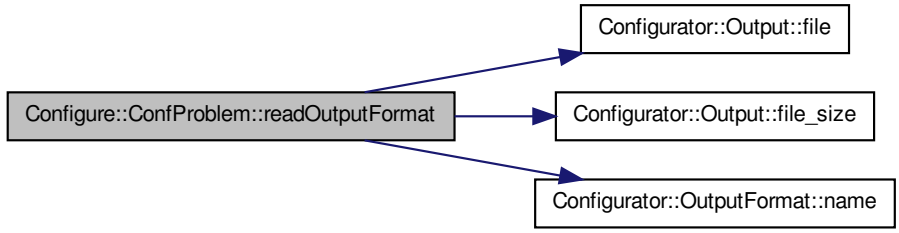
in	*output*	Output structure.

Returns

void

Reads the parameters related to the format of the output files.

Here is the call graph for this function:



void Configure::ConfProblem::readSquare ([Configurator::Cell]cell)

Reads the Square structure.

Parameters

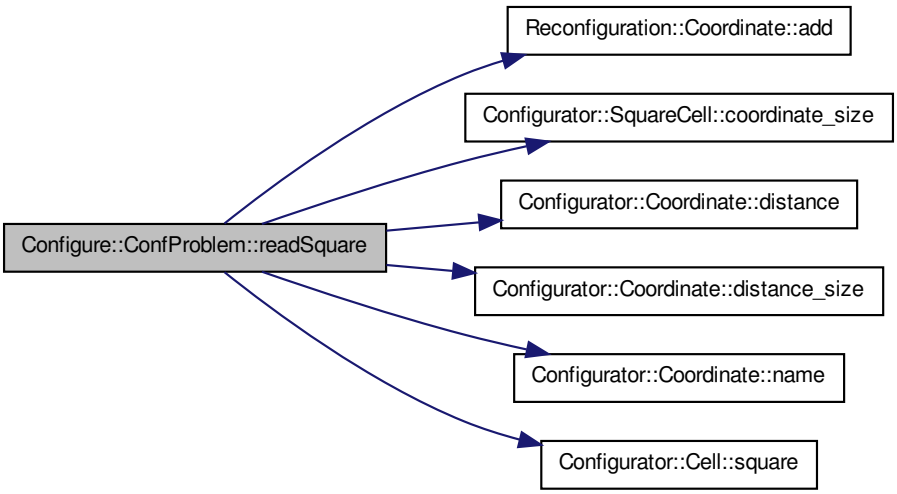
in	*cell*	Cell structure.

Returns

void

Reads the square parameters structure from the configuration file.

Here is the call graph for this function:



void Configure::ConfProblem::readTopology ([Configurator::Graph]graph)

Reads the Topology for a graph structure.

Parameters

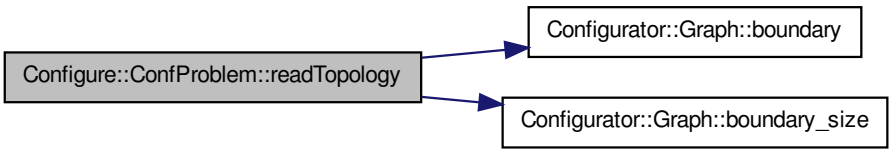
in	*graph*	Graph problem structure.

Returns

void

Reads the parameters of the topology structure if its based on a graph from the configuration file.

Here is the call graph for this function:



void Configure::ConfProblem::readTopology ([Configurator::Simulation]simulation)

Reads the Topology structure.

Parameters

in	*simulation*	Simulation structure.

Returns

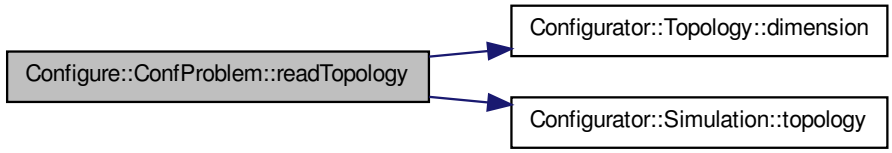
void

Reads the parameters of the topology structure from the configuration file. Acts as a skeleton depending on the topology defined (matrix other graph).

Exceptions

ConfigurationException	Error while reading the configuration `file:` values not recognized, invalid arguments...

Here is the call graph for this function:



void Configure::ConfProblem::readTopology ([Configurator::Matrix]matrix)

Reads the Topology for a matrix structure.

Parameters

`in`	*matrix*	Matrix structure.

Returns

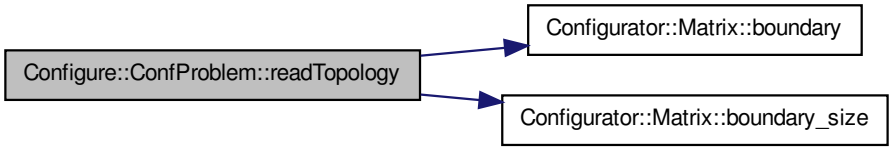
void

Reads the parameters of the topology structure if its based on a matrix from the configuration file.

Exceptions

ConfigurationException	Error while reading the configuration `file:` values not recognized, invalid arguments...

Here is the call graph for this function:



void Configure::ConfProblem::readTriangular ([Configurator::Cell]cell)

Reads the Triangular structure.

Parameters

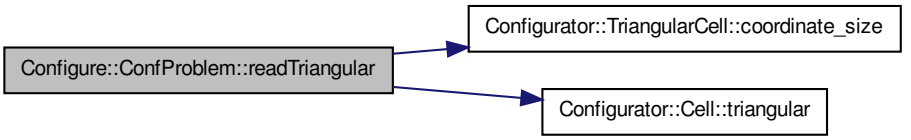
in	*cell*	Cell structure.

Returns

void

Reads the parameters of the triangular structure from the configuration file.

Here is the call graph for this function:



void Configure::ConfProblem::write ( ) [virtual]

Writes the configuration file for the problem specifications.

Returns

void

Writes the configuration file for the problem specifications. The cluster configuration file can be directly written. This is only an example of how to write a configuration file using the platform.

Implements Configure::Configuration.

The documentation for this class was generated from the following files:

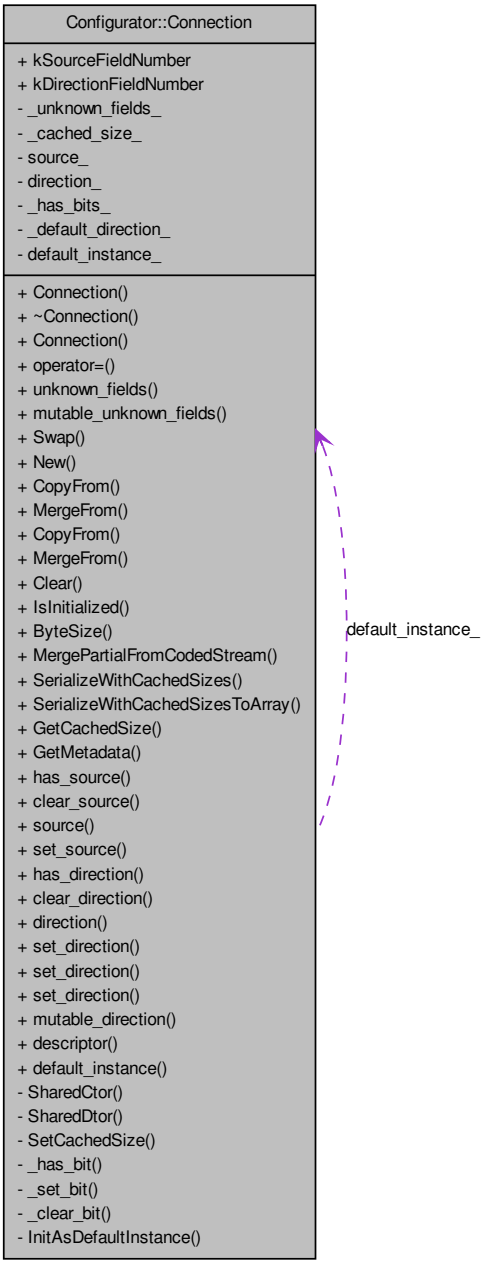
- Configuration.h

- Configuration.cpp

7.14 Configurator::Connection Class Reference

`#include <confproblem.pb.h>`

Collaboration diagram for Configurator::Connection:



7.14.1 Public Member Functions

- Connection ()
- virtual ~Connection ()
- Connection (const Connection &from)
- Connection & operator= (const Connection &from)
- const ::google::protobuf::UnknownFieldSet & unknown_fields () const
- inline::google::protobuf::UnknownFieldSet * mutable_unknown_fields ()
- void Swap (Connection *other)
- Connection * New () const
- void CopyFrom (const ::google::protobuf::Message &from)
- void MergeFrom (const ::google::protobuf::Message &from)
- void CopyFrom (const Connection &from)
- void MergeFrom (const Connection &from)
- void Clear ()
- bool IsInitialized () const
- int ByteSize () const
- bool MergePartialFromCodedStream (::google::protobuf::io::CodedInputStream *input)
- void SerializeWithCachedSizes (::google::protobuf::io::CodedOutputStream *output) const
- ::google::protobuf::uint8 * SerializeWithCachedSizesToArray (::google::protobuf::uint8 *output) const
- int GetCachedSize () const
- ::google::protobuf::Metadata GetMetadata () const
- bool has_source () const
- void clear_source ()
- inline::google::protobuf::int32 source () const
- void set_source (::google::protobuf::int32 value)
- bool has_direction () const
- void clear_direction ()
- const ::std::string & direction () const
- void set_direction (const ::std::string &value)
- void set_direction (const char *value)
- void set_direction (const char *value, size_t size)

- inline::std::string * mutable_direction ()

7.14.2 Static Public Member Functions

- static const ::google::protobuf::Descriptor * descriptor ()
- static const Connection & default_instance ()

7.14.3 Static Public Attributes

- static const int kSourceFieldNumber = 1
- static const int kDirectionFieldNumber = 2

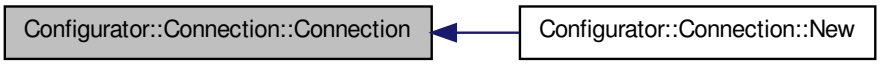
7.14.4 Friends

- void protobuf_AddDesc_confproblem_2eproto ()
- void protobuf_AssignDesc_confproblem_2eproto ()
- void protobuf_ShutdownFile_confproblem_2eproto ()

7.14.5 Constructor & Destructor Documentation

Configurator::Connection::Connection ( )

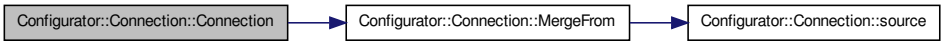
Here is the caller graph for this function:



Configurator::Connection::~Connection ( ) [virtual]

Configurator::Connection::Connection ([const Connection &]from)

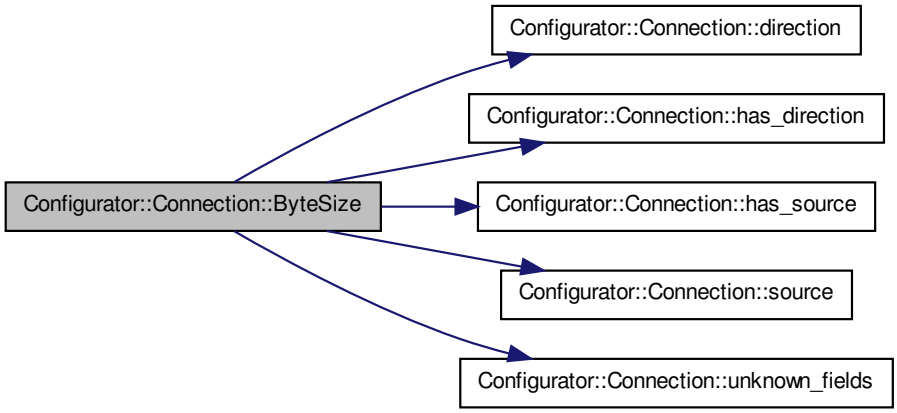
Here is the call graph for this function:



7.14.6 Member Function Documentation

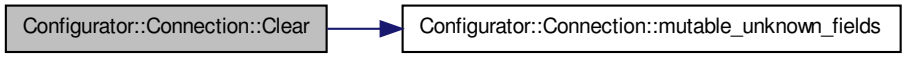
int Configurator::Connection::ByteSize ( ) const

Here is the call graph for this function:

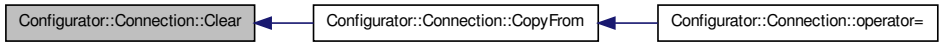


void Configurator::Connection::Clear ( )

Here is the call graph for this function:



Here is the caller graph for this function:

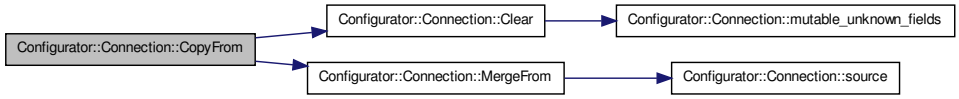


void Configurator::Connection::clear_direction ( ) [inline]

void Configurator::Connection::clear_source ( ) [inline]

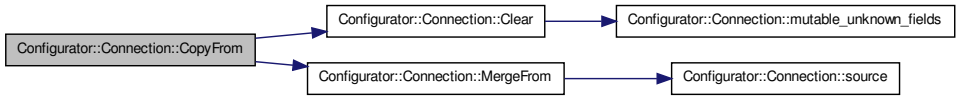
void Configurator::Connection::CopyFrom ([const Connection &]from)

Here is the call graph for this function:



void Configurator::Connection::CopyFrom ([const ::google::protobuf::Message &]from)

Here is the call graph for this function:



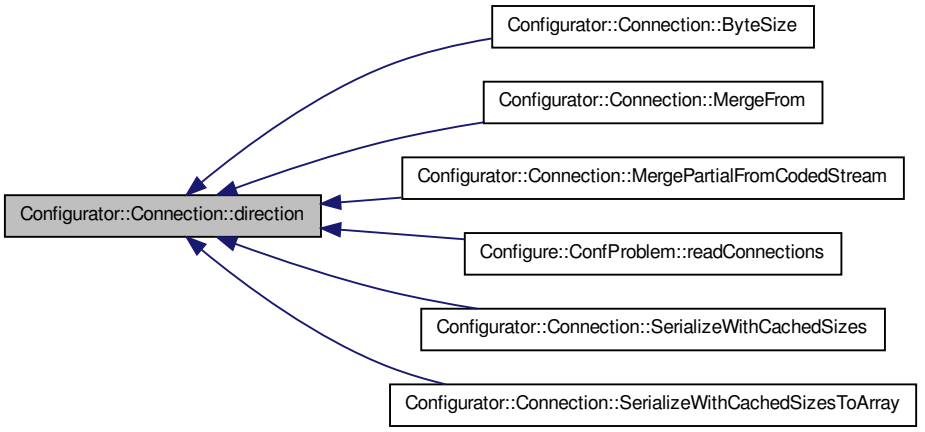
Here is the caller graph for this function:

const Connection & Configurator::Connection::default_instance ( ) [static]

const ::google::protobuf::Descriptor * Configurator::Connection::descriptor ( ) [static]

const ::std::string & Configurator::Connection::direction ( ) const [inline]

Here is the caller graph for this function:

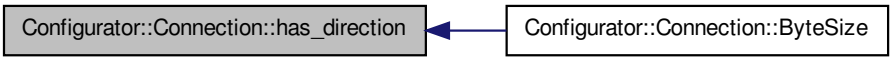


int Configurator::Connection::GetCachedSize ( ) const [inline]

google::protobuf::Metadata Configurator::Connection::GetMetadata ( ) const

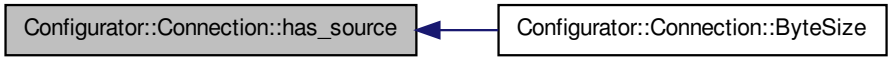
bool Configurator::Connection::has_direction ( ) const [inline]

Here is the caller graph for this function:



bool Configurator::Connection::has_source ( ) const [inline]

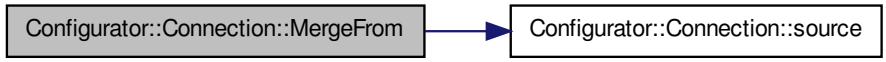
Here is the caller graph for this function:



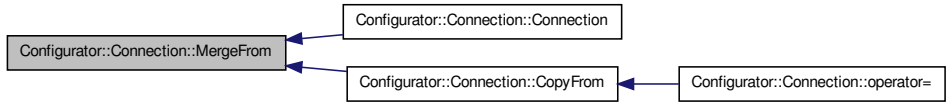
bool Configurator::Connection::IsInitialized ( ) const

void Configurator::Connection::MergeFrom ([const ::google::protobuf::Message &]from)

Here is the call graph for this function:

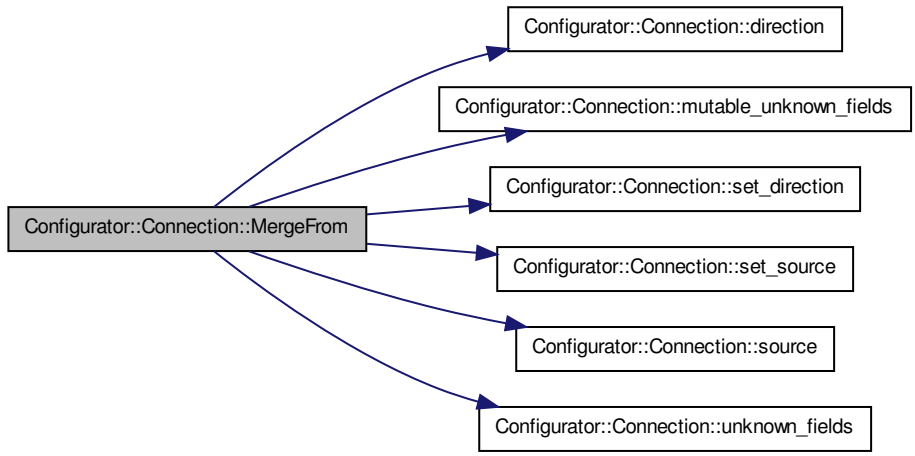


Here is the caller graph for this function:



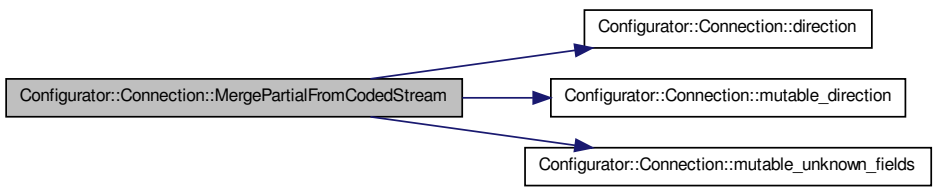
void Configurator::Connection::MergeFrom ([const Connection &]from)

Here is the call graph for this function:



bool Configurator::Connection::MergePartialFromCodedStream ([::google::protobuf::io::CodedInputStream *]input)

Here is the call graph for this function:

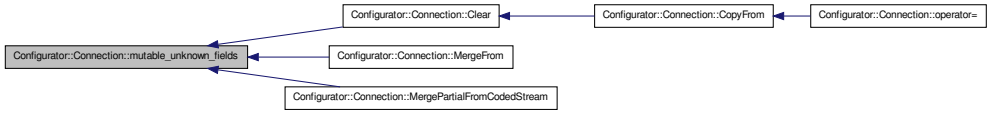


std::string * Configurator::Connection::mutable_direction ( ) [inline]

Here is the caller graph for this function:

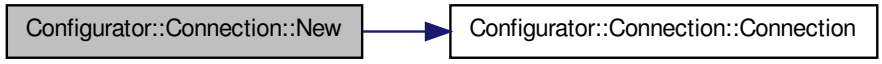
inline ::google::protobuf::UnknownFieldSet* Configurator::Connection::mutable_-unknown_fields ( ) [inline]

Here is the caller graph for this function:



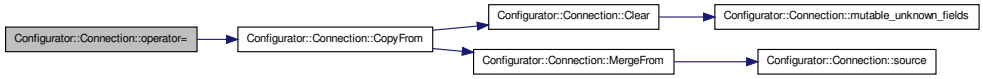
Connection * Configurator::Connection::New ( ) const

Here is the call graph for this function:



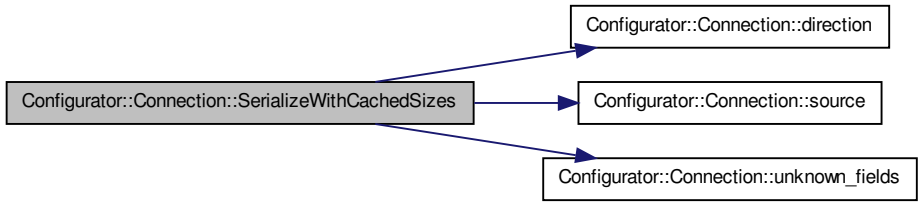
Connection& Configurator::Connection::operator= ([const Connection &]from) [inline]

Here is the call graph for this function:



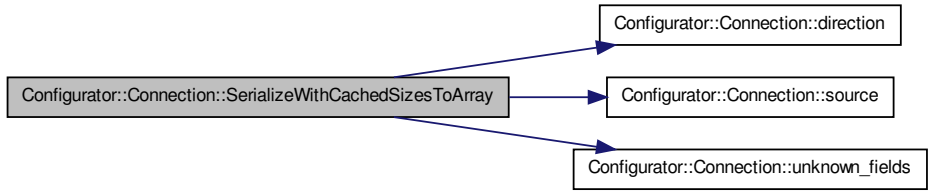
void Configurator::Connection::SerializeWithCachedSizes ([::google::protobuf::io::CodedOutputStream *]output) const

Here is the call graph for this function:



google::protobuf::uint8 * Configurator::Connection::SerializeWithCachedSizesToArray ([::google::protobuf::uint8 *]output) const

Here is the call graph for this function:



void Configurator::Connection::set_direction ([const ::std::string &]value) [inline]

Here is the caller graph for this function:

void Configurator::Connection::set_direction ([const char *]value) [inline]

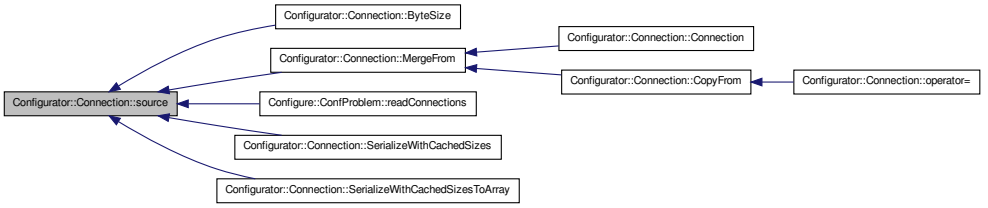
void Configurator::Connection::set_direction ([const char *]value, size_t *size*) [inline]

void Configurator::Connection::set_source ([::google::protobuf::int32]value) [inline]

Here is the caller graph for this function:

google::protobuf::int32 Configurator::Connection::source ( ) const [inline]

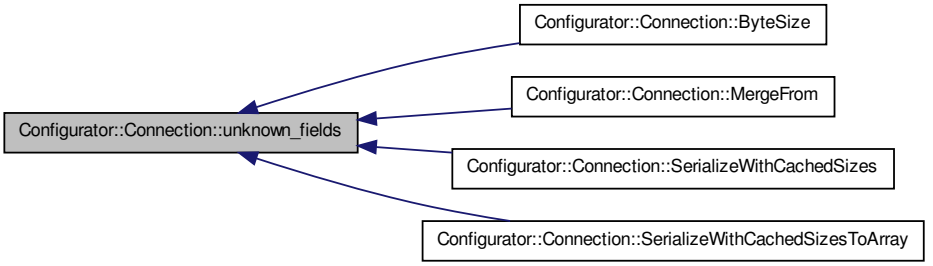
Here is the caller graph for this function:



void Configurator::Connection::Swap ([Connection ∗]other)

const ::google::protobuf::UnknownFieldSet& Configurator::Connection::unknown_-fields ( ) const [inline]

Here is the caller graph for this function:



7.14.7 Friends And Related Function Documentation

void protobuf_AddDesc_confproblem_2eproto ( ) [friend]

void protobuf_AssignDesc_confproblem_2eproto ( ) [friend]

void protobuf_ShutdownFile_confproblem_2eproto ( ) [friend]

7.14.8 Member Data Documentation

const int Configurator::Connection::kDirectionFieldNumber = 2 [static]

const int Configurator::Connection::kSourceFieldNumber = 1 [static]

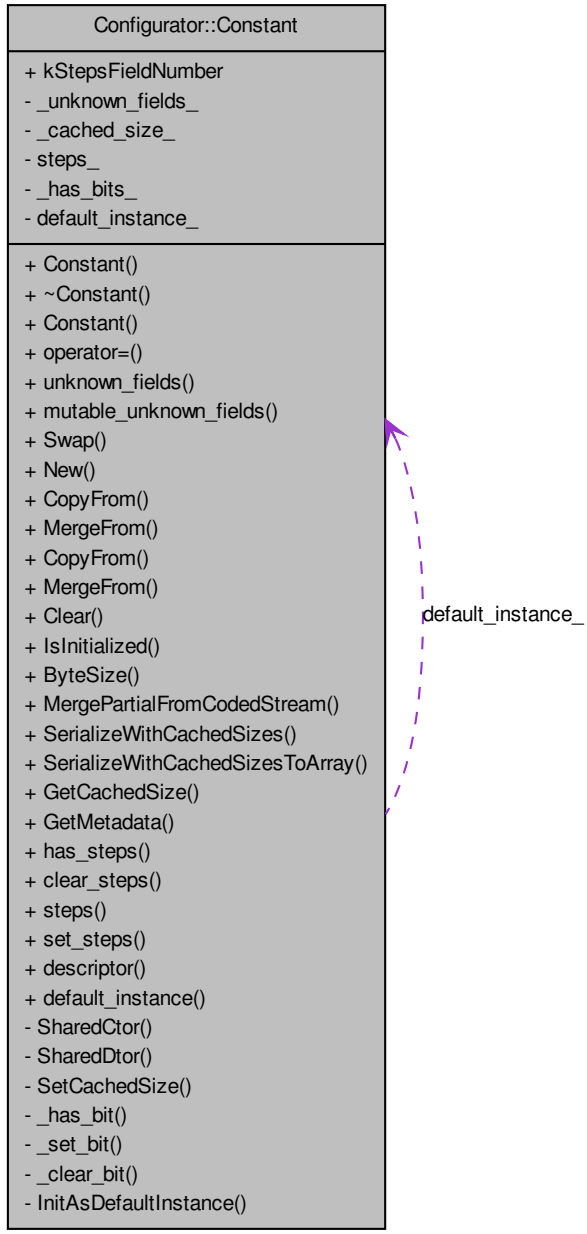
The documentation for this class was generated from the following files:

- confproblem.pb.h
- confproblem.pb.cc

7.15 Configurator::Constant Class Reference

`#include <confpartitioner.pb.h>`

Collaboration diagram for Configurator::Constant:



7.15.1 Public Member Functions

- Constant ()
- virtual ∼Constant ()
- Constant (const Constant &from)
- Constant & operator= (const Constant &from)
- const ::google::protobuf::UnknownFieldSet & unknown_fields () const
- inline::google::protobuf::UnknownFieldSet ∗ mutable_unknown_fields ()
- void Swap (Constant ∗other)
- Constant ∗ New () const
- void CopyFrom (const ::google::protobuf::Message &from)
- void MergeFrom (const ::google::protobuf::Message &from)
- void CopyFrom (const Constant &from)
- void MergeFrom (const Constant &from)
- void Clear ()
- bool IsInitialized () const
- int ByteSize () const
- bool MergePartialFromCodedStream (::google::protobuf::io::CodedInputStream ∗input)
- void SerializeWithCachedSizes (::google::protobuf::io::CodedOutputStream ∗output) const
- ::google::protobuf::uint8 ∗ SerializeWithCachedSizesToArray (::google::protobuf::uint8 ∗output) const
- int GetCachedSize () const
- ::google::protobuf::Metadata GetMetadata () const
- bool has_steps () const
- void clear_steps ()
- inline::google::protobuf::int32 steps () const
- void set_steps (::google::protobuf::int32 value)

7.15.2 Static Public Member Functions

- static const ::google::protobuf::Descriptor ∗ descriptor ()
- static const Constant & default_instance ()

7.15.3 Static Public Attributes

- static const int kStepsFieldNumber = 1

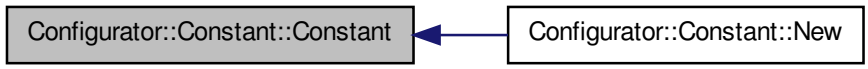
7.15.4 Friends

- void protobuf_AddDesc_confpartitioner_2eproto ()
- void protobuf_AssignDesc_confpartitioner_2eproto ()
- void protobuf_ShutdownFile_confpartitioner_2eproto ()

7.15.5 Constructor & Destructor Documentation

Configurator::Constant::Constant ( )

Here is the caller graph for this function:



Configurator::Constant::~Constant ( ) `[virtual]`

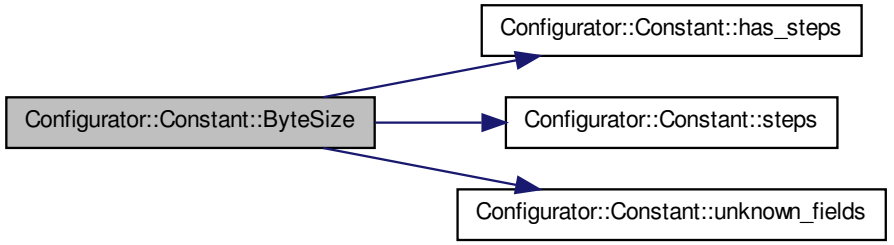
Configurator::Constant::Constant ([const Constant &]from)

Here is the call graph for this function:

7.15.6 Member Function Documentation

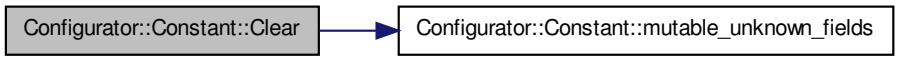
int Configurator::Constant::ByteSize ( ) const

Here is the call graph for this function:

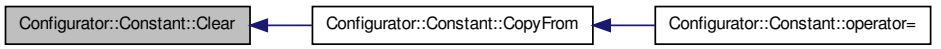


void Configurator::Constant::Clear ( )

Here is the call graph for this function:



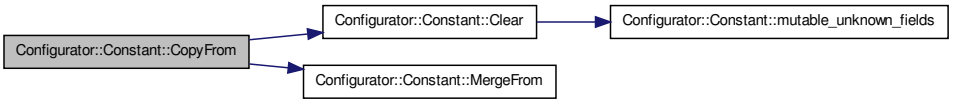
Here is the caller graph for this function:



void Configurator::Constant::clear_steps ( ) `[inline]`

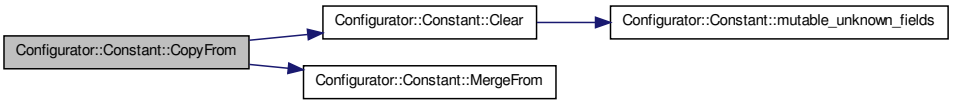
void Configurator::Constant::CopyFrom ([const Constant &]from)

Here is the call graph for this function:



void Configurator::Constant::CopyFrom ([const ::google::protobuf::Message &]from)

Here is the call graph for this function:



Here is the caller graph for this function:

const Constant & Configurator::Constant::default_instance ( ) [static]

const ::google::protobuf::Descriptor * Configurator::Constant::descriptor ( ) [static]

int Configurator::Constant::GetCachedSize ( ) const [inline]

google::protobuf::Metadata Configurator::Constant::GetMetadata ( ) const

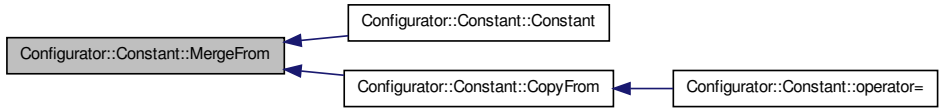
bool Configurator::Constant::has_steps ( ) const [inline]

Here is the caller graph for this function:

bool Configurator::Constant::IsInitialized ( ) const

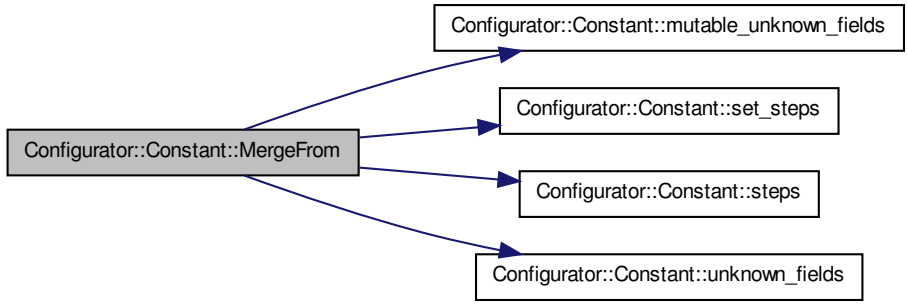
void Configurator::Constant::MergeFrom ([const ::google::protobuf::Message &]from)

Here is the caller graph for this function:



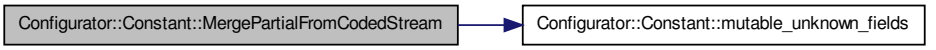
void Configurator::Constant::MergeFrom ([const Constant &]from)

Here is the call graph for this function:



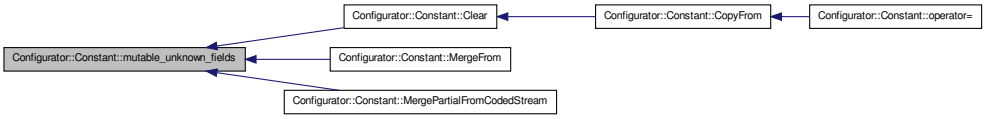
bool Configurator::Constant::MergePartialFromCodedStream ([::google::protobuf::io::CodedInputStream *]input)

Here is the call graph for this function:



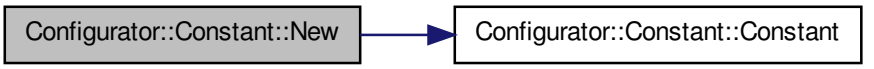
inline ::google::protobuf::UnknownFieldSet* Configurator::Constant::mutable_-unknown_fields ( ) [inline]

Here is the caller graph for this function:



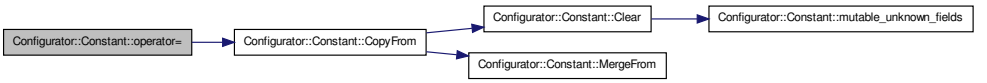
Constant * Configurator::Constant::New ( ) const

Here is the call graph for this function:



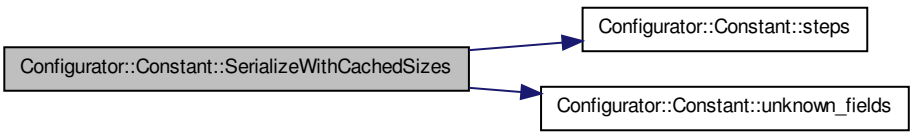
Constant& Configurator::Constant::operator= ([const Constant &]from) [inline]

Here is the call graph for this function:



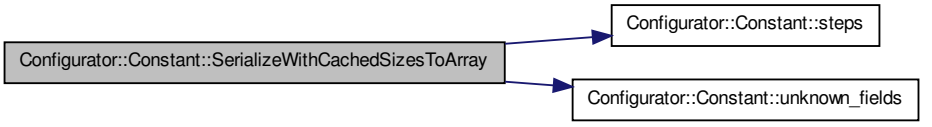
void Configurator::Constant::SerializeWithCachedSizes ([::google::protobuf::io::CodedOutputStream *]output) const

Here is the call graph for this function:



google::protobuf::uint8 * Configurator::Constant::SerializeWithCachedSizesToArray ([::google::protobuf::uint8 *]output) const

Here is the call graph for this function:

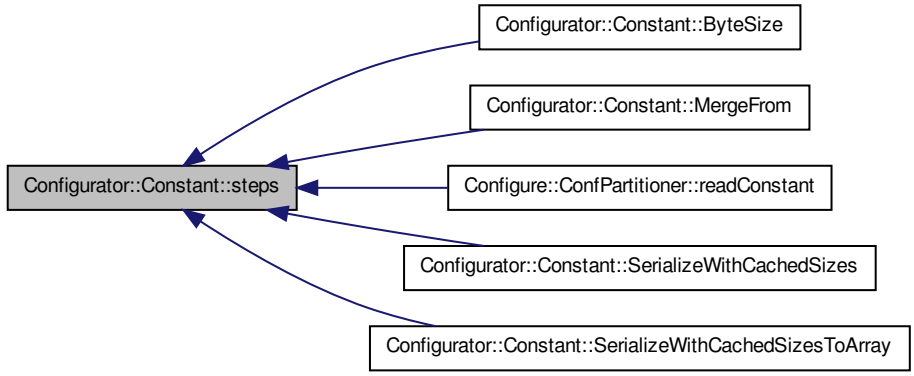


void Configurator::Constant::set_steps ([::google::protobuf::int32]value) [inline]

Here is the caller graph for this function:

google::protobuf::int32 Configurator::Constant::steps ( ) const [inline]

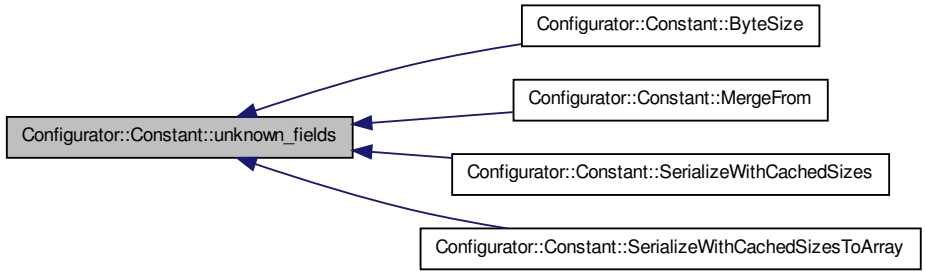
Here is the caller graph for this function:



void Configurator::Constant::Swap ([Constant *]other)

const ::google::protobuf::UnknownFieldSet& Configurator::Constant::unknown_fields ( ) const [inline]

Here is the caller graph for this function:



7.15.7 Friends And Related Function Documentation

void protobuf_AddDesc_confpartitioner_2eproto ( ) [friend]

void protobuf_AssignDesc_confpartitioner_2eproto ( ) [friend]

void protobuf_ShutdownFile_confpartitioner_2eproto ( ) [friend]

7.15.8 Member Data Documentation

const int Configurator::Constant::kStepsFieldNumber = 1 [static]

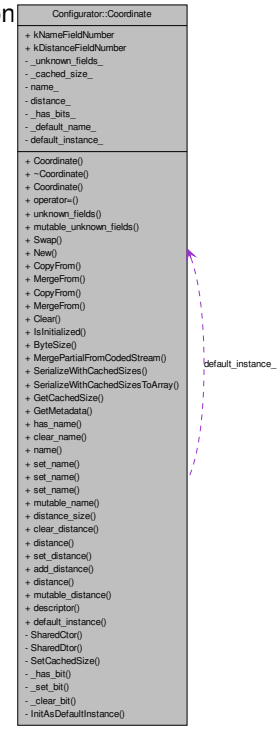
The documentation for this class was generated from the following files:

- confpartitioner.pb.h
- confpartitioner.pb.cc

7.16 Configurator::Coordinate Class Reference

`#include <confproblem.pb.h>`

Collaboration diagram for Con



7.16.1 Public Member Functions

- Coordinate ()
- virtual ∼Coordinate ()
- Coordinate (const Coordinate &from)
- Coordinate & operator= (const Coordinate &from)
- const ::google::protobuf::UnknownFieldSet & unknown_fields () const
- inline::google::protobuf::UnknownFieldSet ∗ mutable_unknown_fields ()
- void Swap (Coordinate ∗other)
- Coordinate ∗ New () const
- void CopyFrom (const ::google::protobuf::Message &from)
- void MergeFrom (const ::google::protobuf::Message &from)
- void CopyFrom (const Coordinate &from)
- void MergeFrom (const Coordinate &from)

- void Clear ()
- bool IsInitialized () const
- int ByteSize () const
- bool MergePartialFromCodedStream (::google::protobuf::io::CodedInputStream ∗input)
- void SerializeWithCachedSizes (::google::protobuf::io::CodedOutputStream ∗output) const
- ::google::protobuf::uint8 ∗ SerializeWithCachedSizesToArray (::google::protobuf::uint8 ∗output) const
- int GetCachedSize () const
- ::google::protobuf::Metadata GetMetadata () const
- bool has_name () const
- void clear_name ()
- const ::std::string & name () const
- void set_name (const ::std::string &value)
- void set_name (const char ∗value)
- void set_name (const char ∗value, size_t size)
- inline::std::string ∗ mutable_name ()
- int distance_size () const
- void clear_distance ()
- inline::google::protobuf::int32 distance (int index) const
- void set_distance (int index,::google::protobuf::int32 value)
- void add_distance (::google::protobuf::int32 value)
- const ::google::protobuf::RepeatedField< ::google::protobuf::int32 > & distance () const
- inline::google::protobuf::RepeatedField< ::google::protobuf::int32 > ∗ mutable_distance ()

7.16.2 Static Public Member Functions

- static const ::google::protobuf::Descriptor ∗ descriptor ()
- static const Coordinate & default_instance ()

7.16.3 Static Public Attributes

- static const int kNameFieldNumber = 1
- static const int kDistanceFieldNumber = 2

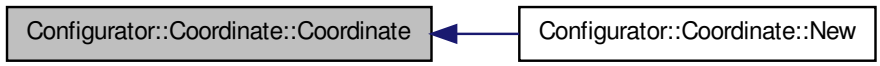
7.16.4 Friends

- void protobuf_AddDesc_confproblem_2eproto ()
- void protobuf_AssignDesc_confproblem_2eproto ()
- void protobuf_ShutdownFile_confproblem_2eproto ()

7.16.5 Constructor & Destructor Documentation

Configurator::Coordinate::Coordinate ( )

Here is the caller graph for this function:



Configurator::Coordinate::~Coordinate ( ) `[virtual]`

Configurator::Coordinate::Coordinate ([const Coordinate &]from)

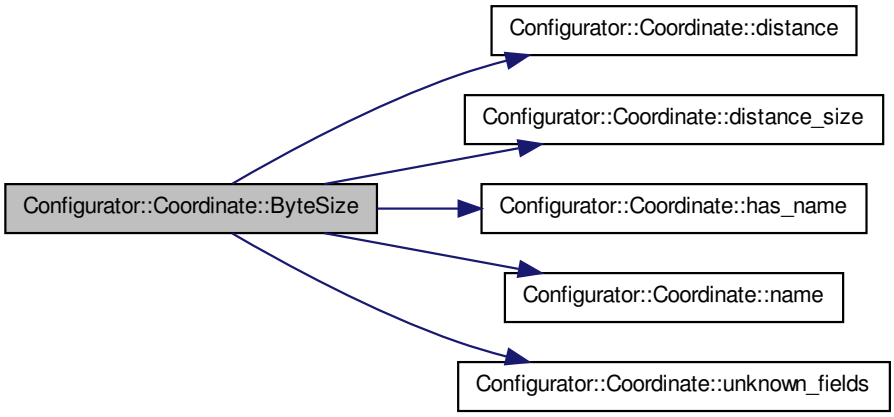
Here is the call graph for this function:

7.16.6 Member Function Documentation

void Configurator::Coordinate::add_distance ([::google::protobuf::int32]value) `[inline]`

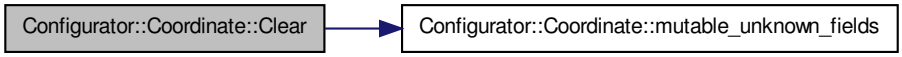
int Configurator::Coordinate::ByteSize ( ) const

Here is the call graph for this function:

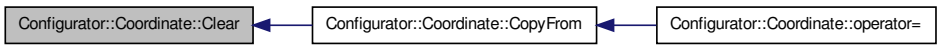


void Configurator::Coordinate::Clear ( )

Here is the call graph for this function:



Here is the caller graph for this function:

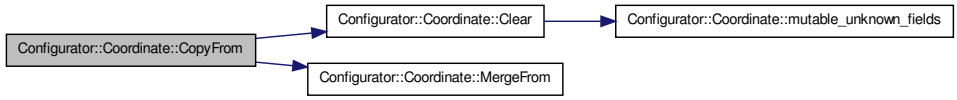


void Configurator::Coordinate::clear_distance ( ) [inline]

void Configurator::Coordinate::clear_name ( ) [inline]

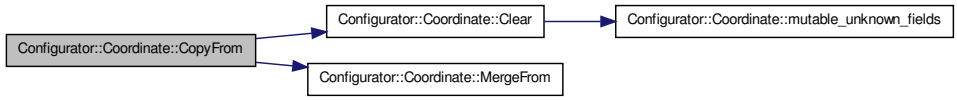
void Configurator::Coordinate::CopyFrom ([const Coordinate &]from)

Here is the call graph for this function:



void Configurator::Coordinate::CopyFrom ([const ::google::protobuf::Message &]from)

Here is the call graph for this function:



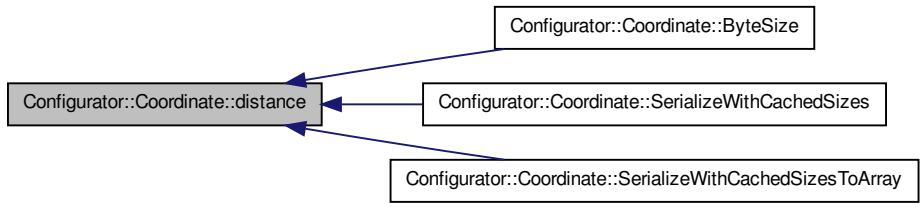
Here is the caller graph for this function:

const Coordinate & Configurator::Coordinate::default_instance ( ) [static]

const ::google::protobuf::Descriptor * Configurator::Coordinate::descriptor ( ) [static]

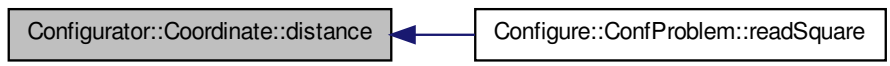
const ::google::protobuf::RepeatedField<::google::protobuf::int32 > & Configurator::Coordinate::distance ( ) const [inline]

Here is the caller graph for this function:



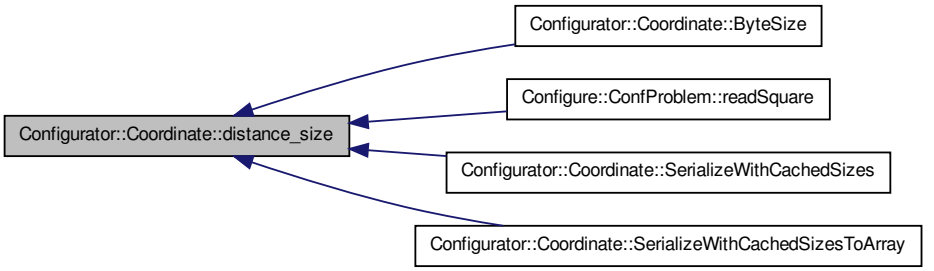
google::protobuf::int32 Configurator::Coordinate::distance ([int]index) const [inline]

Here is the caller graph for this function:



int Configurator::Coordinate::distance_size ( ) const [inline]

Here is the caller graph for this function:



int Configurator::Coordinate::GetCachedSize ( ) const [inline]

google::protobuf::Metadata Configurator::Coordinate::GetMetadata ( ) const

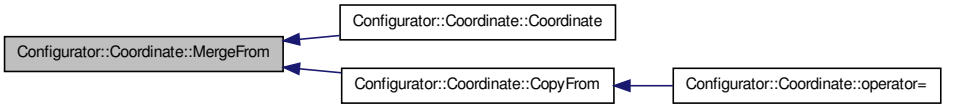
bool Configurator::Coordinate::has_name ( ) const [inline]

Here is the caller graph for this function:

bool Configurator::Coordinate::IsInitialized ( ) const

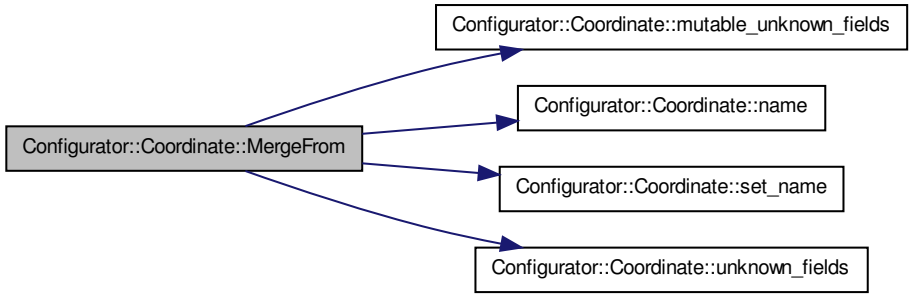
void Configurator::Coordinate::MergeFrom ([const ::google::protobuf::Message &]from)

Here is the caller graph for this function:



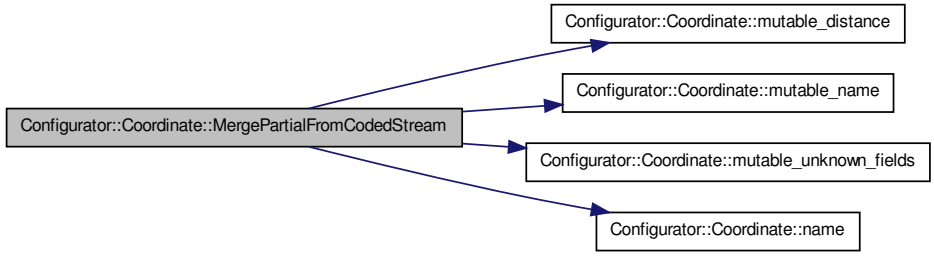
void Configurator::Coordinate::MergeFrom ([const Coordinate &]from)

Here is the call graph for this function:



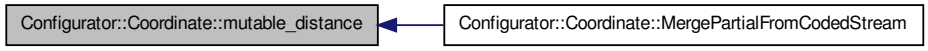
bool Configurator::Coordinate::MergePartialFromCodedStream ([::google::protobuf::io::CodedInputStream *]input)

Here is the call graph for this function:



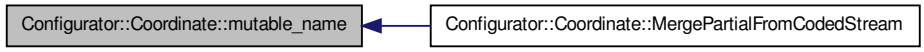
google::protobuf::RepeatedField<::google::protobuf::int32 > * Configurator::Coordinate::mutable_distance ( ) [inline]

Here is the caller graph for this function:



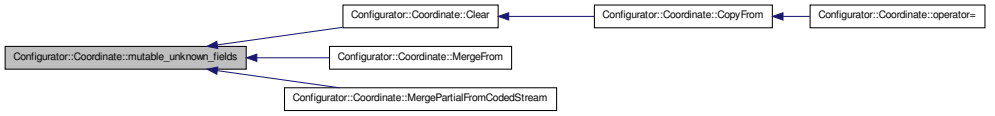
std::string * Configurator::Coordinate::mutable_name ( ) [inline]

Here is the caller graph for this function:



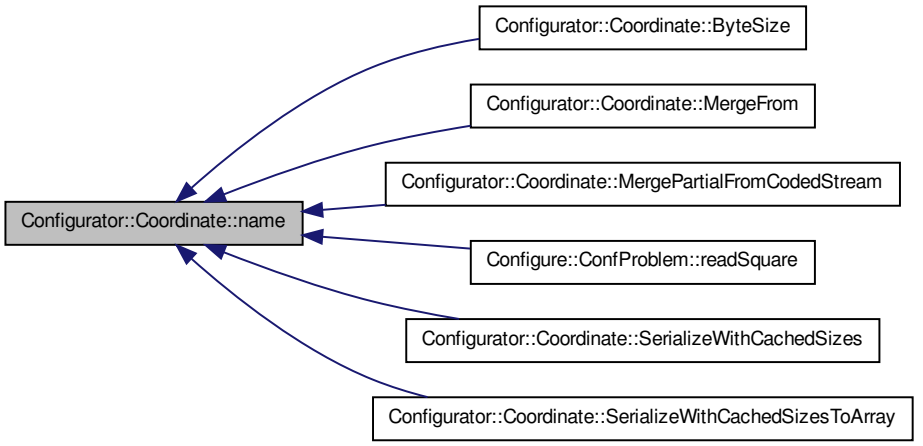
inline ::google::protobuf::UnknownFieldSet* Configurator::Coordinate::mutable_unknown_fields ( ) [inline]

Here is the caller graph for this function:



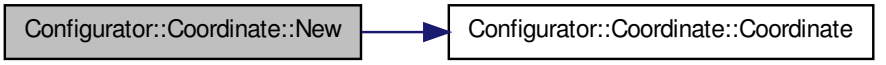
const ::std::string & Configurator::Coordinate::name ( ) const `[inline]`

Here is the caller graph for this function:



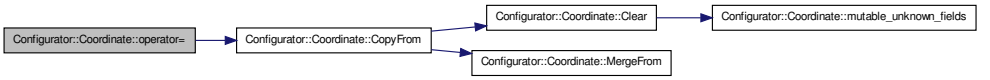
Coordinate * Configurator::Coordinate::New ( ) const

Here is the call graph for this function:



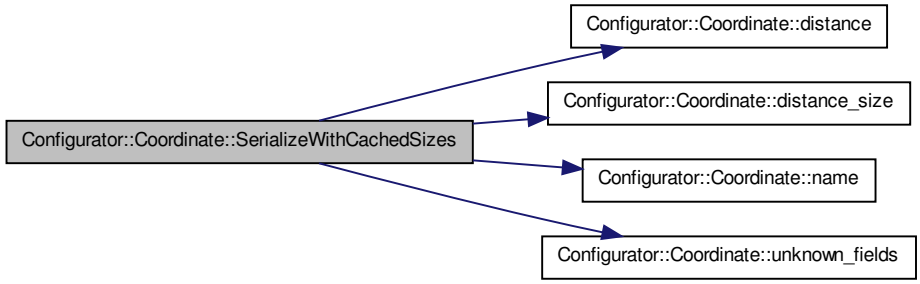
Coordinate& Configurator::Coordinate::operator= ([const Coordinate &]from) `[inline]`

Here is the call graph for this function:



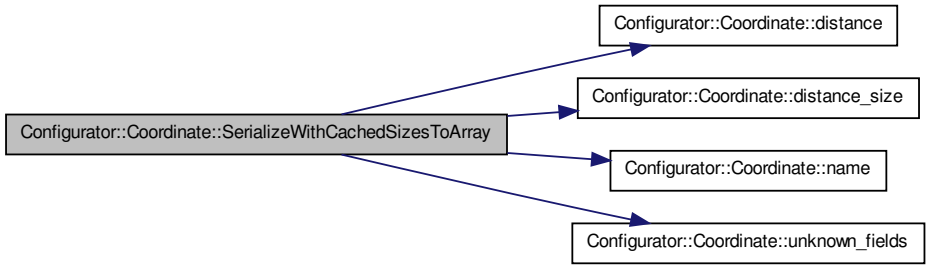
void Configurator::Coordinate::SerializeWithCachedSizes ([::google::protobuf::io::CodedOutputStream *]output) const

Here is the call graph for this function:



google::protobuf::uint8 * Configurator::Coordinate::SerializeWithCachedSizesToArray ([::google::protobuf::uint8 *]output) const

Here is the call graph for this function:



void Configurator::Coordinate::set_distance ([int]index, ::google::protobuf::int32 *value*) [inline]

void Configurator::Coordinate::set_name ([const ::std::string &]value) [inline]

Here is the caller graph for this function:

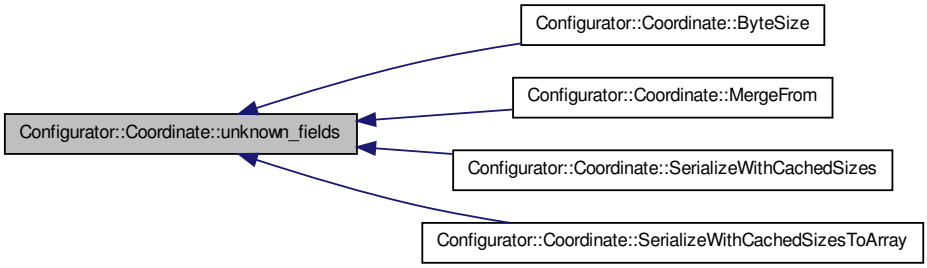
void Configurator::Coordinate::set_name ([const char *]value) [inline]

void Configurator::Coordinate::set_name ([const char *]value, size_t *size*) [inline]

void Configurator::Coordinate::Swap ([Coordinate *]other)

const ::google::protobuf::UnknownFieldSet& Configurator::Coordinate::unknown_fields ( ) const [inline]

Here is the caller graph for this function:



7.16.7 Friends And Related Function Documentation

void protobuf_AddDesc_confproblem_2eproto ( ) [friend]

void protobuf_AssignDesc_confproblem_2eproto ( ) [friend]

void protobuf_ShutdownFile_confproblem_2eproto ( ) [friend]

7.16.8 Member Data Documentation

const int Configurator::Coordinate::kDistanceFieldNumber = 2 [static]

const int Configurator::Coordinate::kNameFieldNumber = 1 [static]

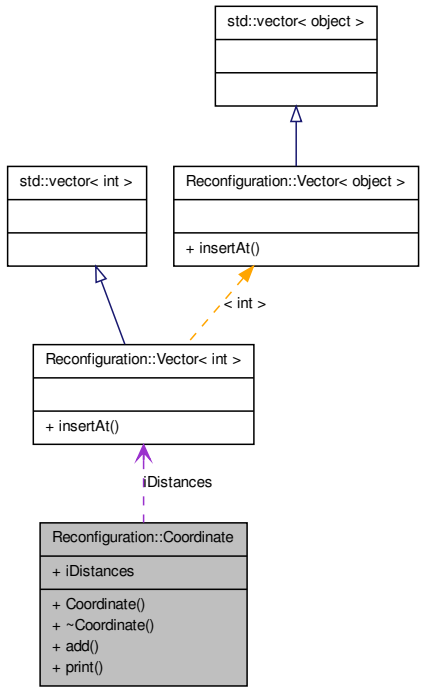
The documentation for this class was generated from the following files:

- confproblem.pb.h
- confproblem.pb.cc

7.17 Reconfiguration::Coordinate Class Reference

`#include <Coordinate.h>`

Collaboration diagram for Reconfiguration::Coordinate:



7.17.1 Public Member Functions

- Coordinate ()

  Constructor of the Coordinate class.

- virtual ~Coordinate ()

  virtual destructor of the class Coordinate.

- void add (int)

  Adds a new integer value to the coordinate.

- void print ()

  Prints a coordinate object.

7.17.2 Public Attributes

- Vector< int > iDistances

7.17.3 Detailed Description

Set of integer values to define the direction of the links at the configuration file $problem.conf written by the user.

7.17.4 Constructor & Destructor Documentation

Reconfiguration::Coordinate::Coordinate ( )

Constructor of the Coordinate class.

Returns

∗this

Creates an object Coordinate.

Reconfiguration::Coordinate::∼Coordinate ( ) `[virtual]`

virtual destructor of the class Coordinate.

Returns

void

Destroys the Coordinate object.

7.17.5 Member Function Documentation

void Reconfiguration::Coordinate::add ([int]iValue)

Adds a new integer value to the coordinate.

Parameters

`in`	*iValue*	Value to be added.

Returns

void

Adds a new integer value to the coordinate, that corresponds with the next dimension readed. The new value is added to the vector following a stack policy (push_back).

Here is the caller graph for this function:

void Reconfiguration::Coordinate::print ( )

Prints a coordinate object.

Returns

void

Prints the vector elements of the coordinate.

7.17.6 Member Data Documentation

Vector< int > Reconfiguration::Coordinate::iDistances

Vector of integer values to store each coordenate of the direction of the link (data dependency).

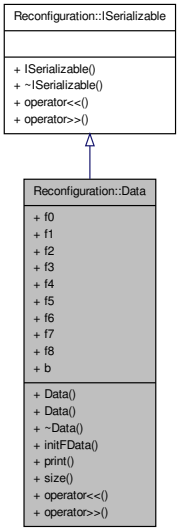
The documentation for this class was generated from the following files:

- Coordinate.h
- Coordinate.cpp

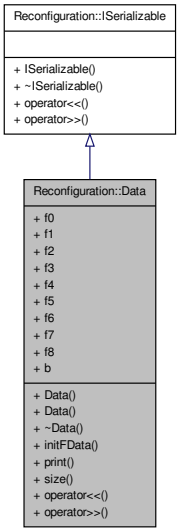
7.18 Reconfiguration::Data Class Reference

`#include <Data.h>`

Inheritance diagram for Reconfiguration::Data:



Collaboration diagram for Reconfiguration::Data:



7.18.1 Public Member Functions

- Data ()

- Data (double)
- virtual ∼Data ()
- void initFData (double)
- void print ()
- int size ()
- std::stringstream & operator<< (std::stringstream &)

  Deserialization of an object.

- std::stringstream & operator>> (std::stringstream &)

  Serialization of an object.

7.18.2 Public Attributes

- double f0
- double f1
- double f2
- double f3
- double f4
- double f5
- double f6
- double f7
- double f8
- double b

7.18.3 Constructor & Destructor Documentation

Reconfiguration::Data::Data ( )

Reconfiguration::Data::Data ([double])

virtual Reconfiguration::Data::∼Data ( ) `[virtual]`

7.18.4 Member Function Documentation

void Reconfiguration::Data::initFData ([double])

std::stringstream& Reconfiguration::Data::operator<< ([std::stringstream &]) `[virtual]`

Deserialization of an object.

Returns

std::stringstream

Deserialization of an object. Returns the modified std::stringstream. Implemented by user in the Data file.

Implements Reconfiguration::ISerializable.

std::stringstream& Reconfiguration::Data::operator>> ([std::stringstream &]) `[virtual]`

Serialization of an object.

Returns

std::stringstream

Serialization of an object. Returns the modified std::stringstream. Implemented by user in the Data file.

Implements Reconfiguration::ISerializable.

void Reconfiguration::Data::print ( )

Here is the caller graph for this function:

int Reconfiguration::Data::size ( )

7.18.5 Member Data Documentation

double Reconfiguration::Data::b

double Reconfiguration::Data::f0

double Reconfiguration::Data::f1

double Reconfiguration::Data::f2

double Reconfiguration::Data::f3

double Reconfiguration::Data::f4

double Reconfiguration::Data::f5

double Reconfiguration::Data::f6

double Reconfiguration::Data::f7

double Reconfiguration::Data::f8

The documentation for this class was generated from the following file:

- Data.h

7.19 Reconfiguration::Framework Class Reference

`#include <Framework.h>`

7.19.1 Public Member Functions

- Framework (int, char $**$, Algorithm $*$)

  Constructor of the Framework class.

- virtual $\sim$Framework ()

  virtual destructor of the class Framework.

7.19.2 Detailed Description

Class is in charge of creating and managing the master and slave processes that are going to be executed in parallel.

7.19.3 Constructor & Destructor Documentation

Reconfiguration::Framework::Framework ([int], char $**$, Algorithm $*$)

Constructor of the Framework class.

Parameters

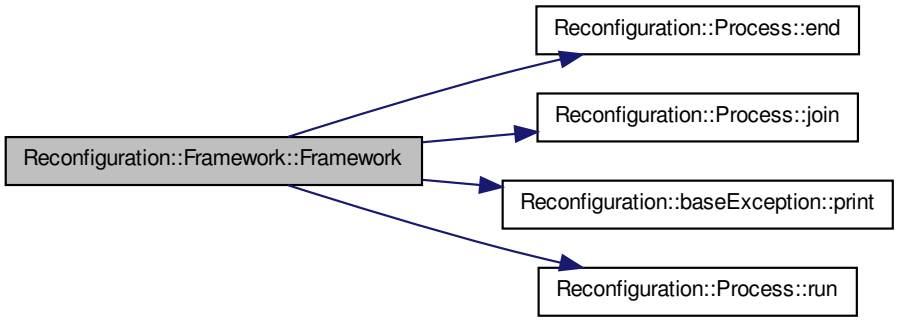
in	*argc*	arguments counter (argc).
in	*argv*	vector of arguments ($*$argv[]).
in	*algorithm*	pointer to the Algorithm object.

Returns

$*$this

Creates an object Framework by initializing the MPI layer, the MPI error layer and creates the master and slave processes. It also calls the join, run and end methods of the processes.

Here is the call graph for this function:



Reconfiguration::Framework::~Framework ( ) `[virtual]`

virtual destructor of the class Framework.

Returns

void

Destroys the Framework object by finalizing the MPI layer.

Returns

void

Destroys the Framework object finalizing the MPI layer.

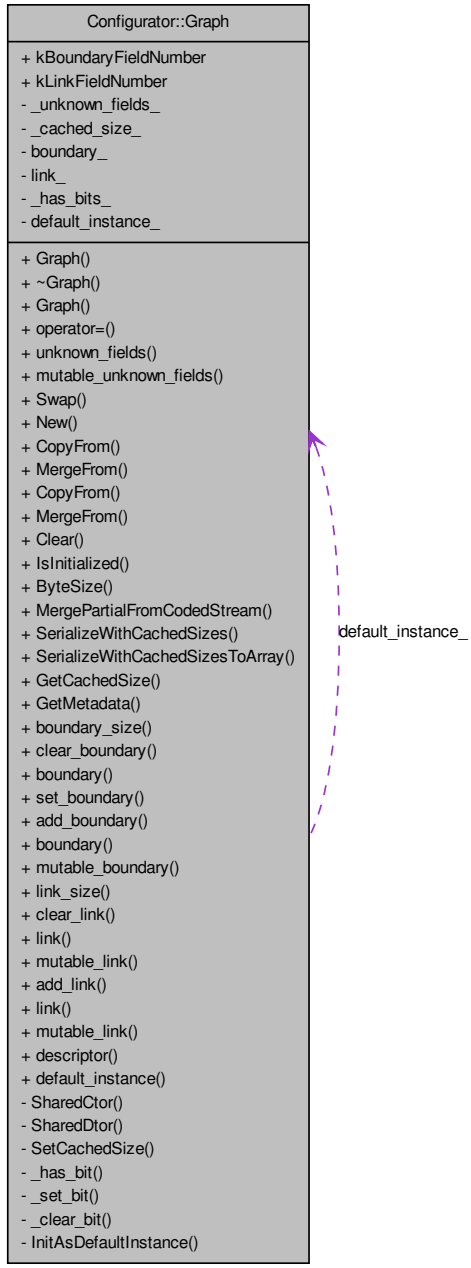
The documentation for this class was generated from the following files:

- Framework.h
- Framework.cpp

7.20 Configurator::Graph Class Reference

`#include <confproblem.pb.h>`

Collaboration diagram for Configurator::Graph:



7.20.1 Public Member Functions

- Graph ()
- virtual ∼Graph ()
- Graph (const Graph &from)
- Graph & operator= (const Graph &from)
- const ::google::protobuf::UnknownFieldSet & unknown_fields () const
- inline::google::protobuf::UnknownFieldSet ∗ mutable_unknown_fields ()
- void Swap (Graph ∗other)
- Graph ∗ New () const
- void CopyFrom (const ::google::protobuf::Message &from)
- void MergeFrom (const ::google::protobuf::Message &from)
- void CopyFrom (const Graph &from)
- void MergeFrom (const Graph &from)
- void Clear ()
- bool IsInitialized () const
- int ByteSize () const
- bool MergePartialFromCodedStream (::google::protobuf::io::CodedInputStream ∗input)
- void SerializeWithCachedSizes (::google::protobuf::io::CodedOutputStream ∗output) const
- ::google::protobuf::uint8 ∗ SerializeWithCachedSizesToArray (::google::protobuf::uint8 ∗output) const
- int GetCachedSize () const
- ::google::protobuf::Metadata GetMetadata () const
- int boundary_size () const
- void clear_boundary ()
- bool boundary (int index) const
- void set_boundary (int index, bool value)
- void add_boundary (bool value)
- const ::google::protobuf::RepeatedField< bool > & boundary () const
- inline::google::protobuf::RepeatedField< bool > ∗ mutable_boundary ()
- int link_size () const
- void clear_link ()
- const ::Configurator::Connection & link (int index) const

- inline::Configurator::Connection * mutable_link (int index)
- inline::Configurator::Connection * add_link ()
- const ::google::protobuf::RepeatedPtrField< ::Configurator::Connection > & link () const
- inline::google::protobuf::RepeatedPtrField< ::Configurator::Connection > * mutable_link ()

7.20.2 Static Public Member Functions

- static const ::google::protobuf::Descriptor * descriptor ()
- static const Graph & default_instance ()

7.20.3 Static Public Attributes

- static const int kBoundaryFieldNumber = 1
- static const int kLinkFieldNumber = 2

7.20.4 Friends

- void protobuf_AddDesc_confproblem_2eproto ()
- void protobuf_AssignDesc_confproblem_2eproto ()
- void protobuf_ShutdownFile_confproblem_2eproto ()

7.20.5 Constructor & Destructor Documentation

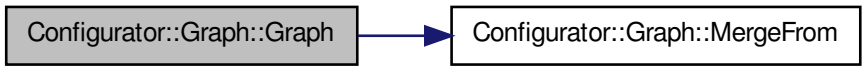
Configurator::Graph::Graph ( )

Here is the caller graph for this function:

Configurator::Graph::~Graph ( ) [virtual]

Configurator::Graph::Graph ([const Graph &]from)

Here is the call graph for this function:

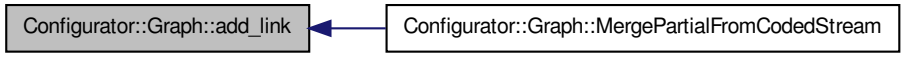


7.20.6 Member Function Documentation

void Configurator::Graph::add_boundary ([bool]value) `[inline]`

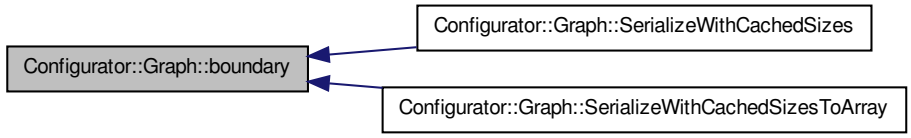
Configurator::Connection * Configurator::Graph::add_link ( ) `[inline]`

Here is the caller graph for this function:



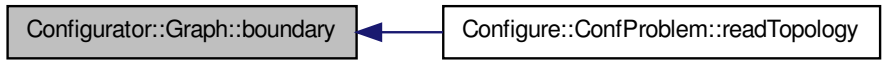
const ::google::protobuf::RepeatedField< bool > & Configurator::Graph::boundary ( ) const `[inline]`

Here is the caller graph for this function:



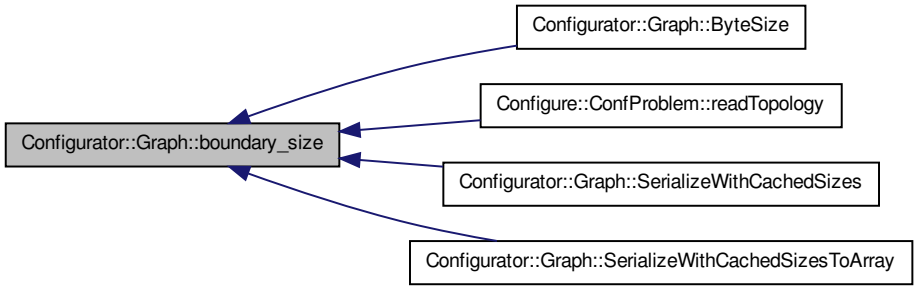
bool Configurator::Graph::boundary ([int]index) const `[inline]`

Here is the caller graph for this function:



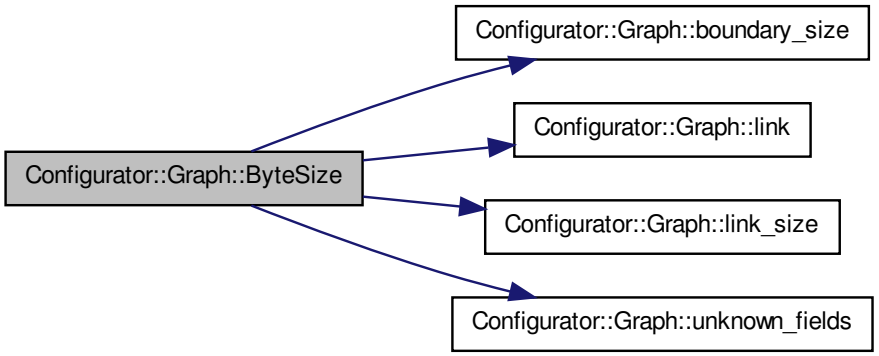
int Configurator::Graph::boundary_size ( ) const `[inline]`

Here is the caller graph for this function:



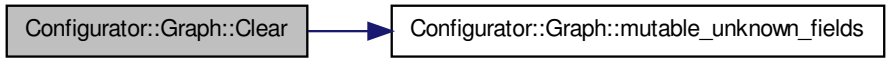
int Configurator::Graph::ByteSize ( ) const

Here is the call graph for this function:

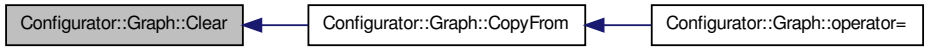


void Configurator::Graph::Clear ( )

Here is the call graph for this function:



Here is the caller graph for this function:

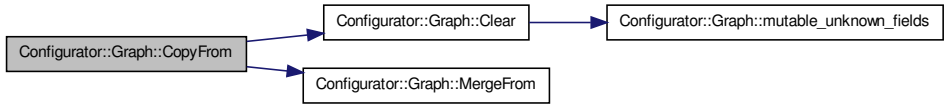


void Configurator::Graph::clear_boundary ( ) [inline]

void Configurator::Graph::clear_link ( ) [inline]

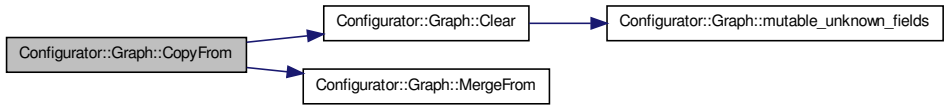
void Configurator::Graph::CopyFrom ([const Graph &]from)

Here is the call graph for this function:



void Configurator::Graph::CopyFrom ([const ::google::protobuf::Message &]from)

Here is the call graph for this function:



Here is the caller graph for this function:

const Graph & Configurator::Graph::default_instance ( ) [static]

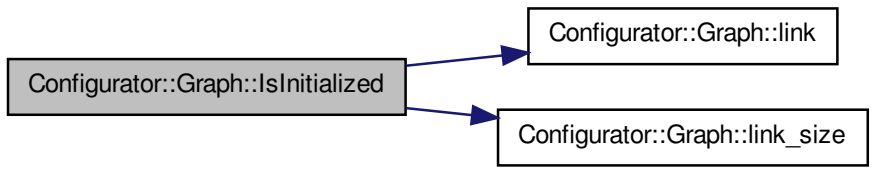
const ::google::protobuf::Descriptor * Configurator::Graph::descriptor ( ) [static]

int Configurator::Graph::GetCachedSize ( ) const [inline]

google::protobuf::Metadata Configurator::Graph::GetMetadata ( ) const

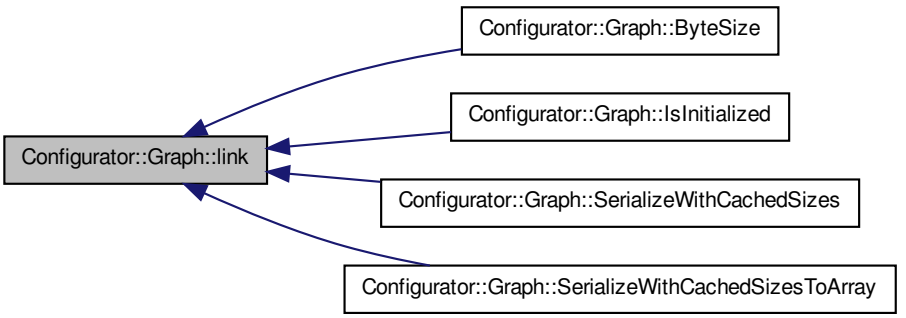
bool Configurator::Graph::IsInitialized ( ) const

Here is the call graph for this function:



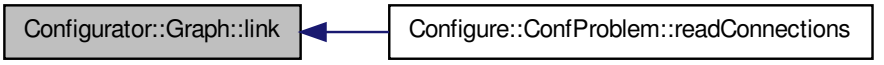
const ::google::protobuf::RepeatedPtrField<::Configurator::Connection > & Configurator::Graph::link ( ) const [inline]

Here is the caller graph for this function:



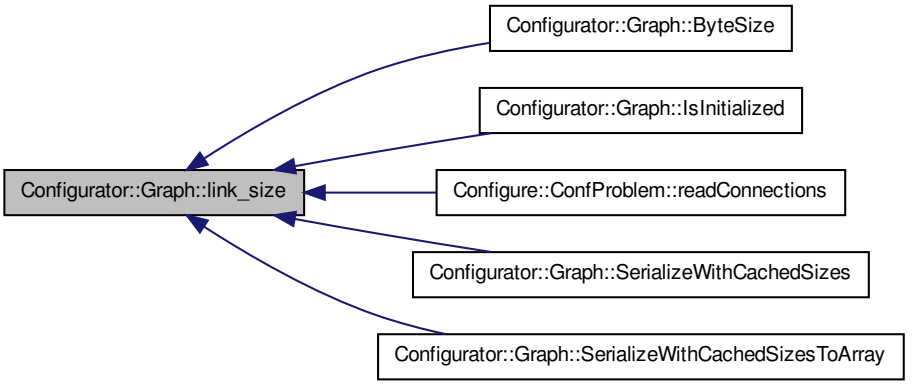
const ::Configurator::Connection & Configurator::Graph::link ([int]index) const `[inline]`

Here is the caller graph for this function:



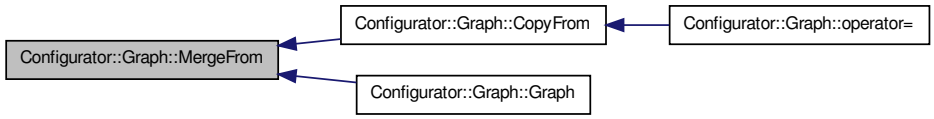
int Configurator::Graph::link_size ( ) const `[inline]`

Here is the caller graph for this function:



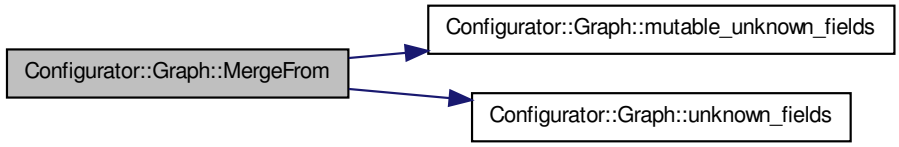
void Configurator::Graph::MergeFrom ([const ::google::protobuf::Message &]from)

Here is the caller graph for this function:



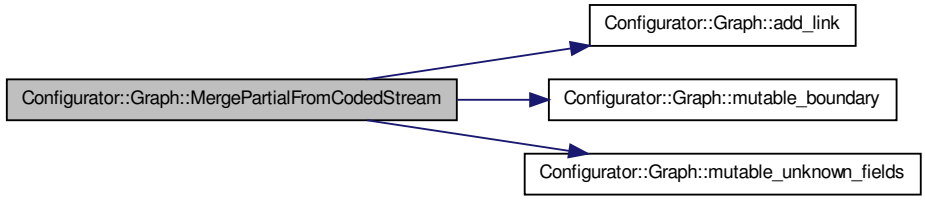
void Configurator::Graph::MergeFrom ([const Graph &]from)

Here is the call graph for this function:



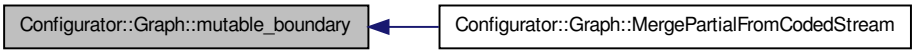
bool Configurator::Graph::MergePartialFromCodedStream ([::google::protobuf::io::CodedInputStream *]input)

Here is the call graph for this function:



google::protobuf::RepeatedField< bool > * Configurator::Graph::mutable_boundary ( ) [inline]

Here is the caller graph for this function:

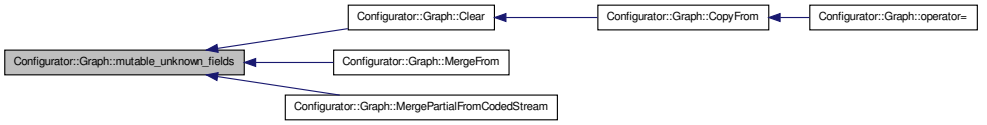


google::protobuf::RepeatedPtrField<::Configurator::Connection > * Configurator::Graph::mutable_link ( ) [inline]

Configurator::Connection * Configurator::Graph::mutable_link ([int]index) [inline]

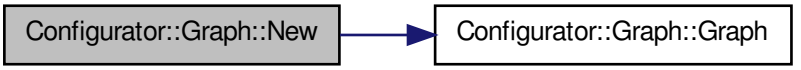
inline ::google::protobuf::UnknownFieldSet* Configurator::Graph::mutable_unknown_fields ( ) [inline]

Here is the caller graph for this function:



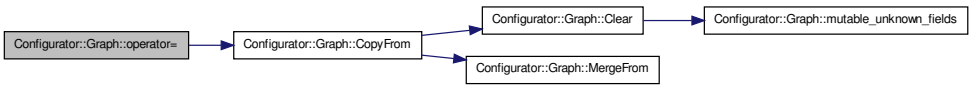
Graph * Configurator::Graph::New ( ) const

Here is the call graph for this function:



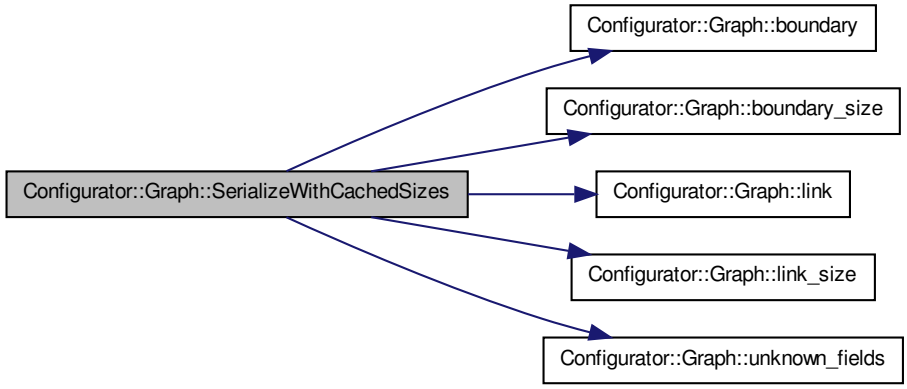
Graph& Configurator::Graph::operator= ([const Graph &]from) [inline]

Here is the call graph for this function:



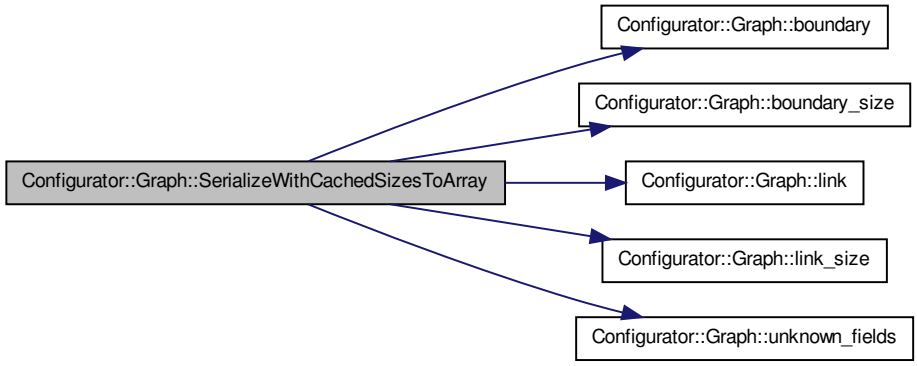
void Configurator::Graph::SerializeWithCachedSizes ([::google::protobuf::io::CodedOutputStream *]output) const

Here is the call graph for this function:



google::protobuf::uint8 * Configurator::Graph::SerializeWithCachedSizesToArray ([::google::protobuf::uint8 *]output) const

Here is the call graph for this function:

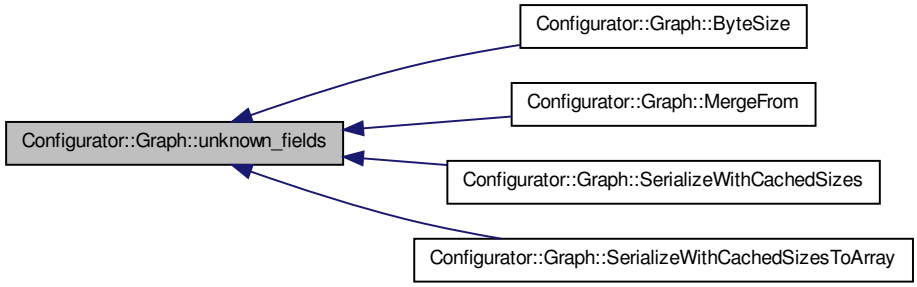


void Configurator::Graph::set_boundary ([int]index, bool *value*) `[inline]`

void Configurator::Graph::Swap ([Graph *]other)

const ::google::protobuf::UnknownFieldSet& Configurator::Graph::unknown_fields ( ) const `[inline]`

Here is the caller graph for this function:



7.20.7 Friends And Related Function Documentation

void protobuf_AddDesc_confproblem_2eproto ( ) [friend]

void protobuf_AssignDesc_confproblem_2eproto ( ) [friend]

void protobuf_ShutdownFile_confproblem_2eproto ( ) [friend]

7.20.8 Member Data Documentation

const int Configurator::Graph::kBoundaryFieldNumber = 1 [static]

const int Configurator::Graph::kLinkFieldNumber = 2 [static]

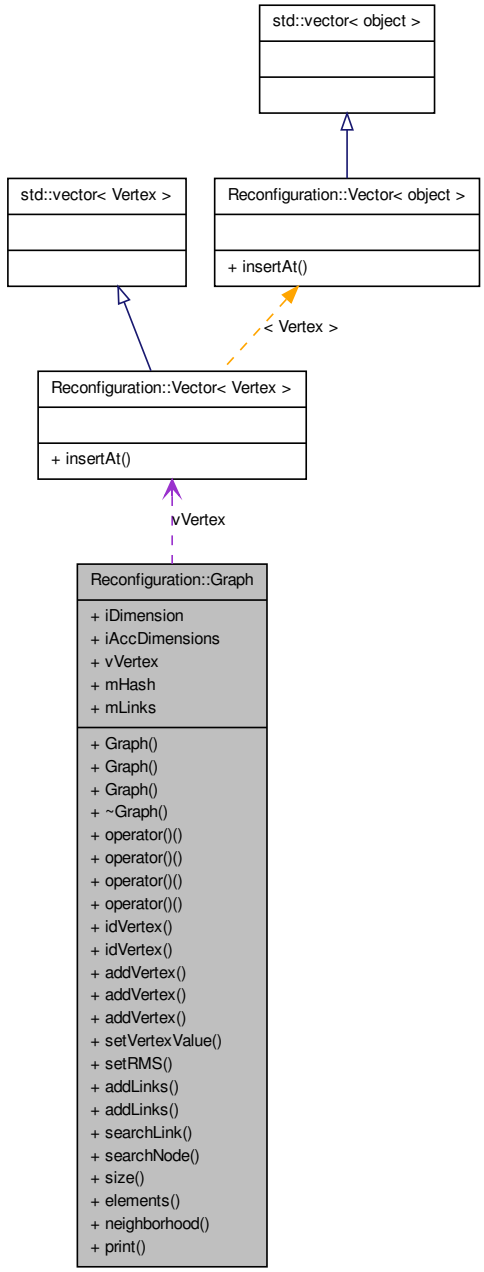
The documentation for this class was generated from the following files:

- confproblem.pb.h
- confproblem.pb.cc

7.21 Reconfiguration::Graph Class Reference

`#include <Graph.h>`

Collaboration diagram for Reconfiguration::Graph:



7.21.1 Public Member Functions

- Graph ()

  Constructor of the Graph class.

- Graph (std::map< std::string, int >, int, int *)

  Constructor of the Graph class.

- Graph (const Graph &)

  Copy constructor of the Graph class.

- virtual ~Graph ()

  virtual destructor of the class Graph.

- Data operator() (int) const

  Access operator ()

- Data & operator() (int)

  Access operator ()

- Data operator() (int, const std::string &)

  Access operator () by linkname.

- Data operator() (int, int *)

  Access operator () by distance vector.

- int idVertex (const std::string &)

  Searches for a certain cluster machine.

- int idVertex (int)

  Searches for a certain graph vertex.

- void addVertex (int, double, double, const std::string &)

  Adds a vertex (machine) to the graph.

- void addVertex (int, Data)

Adds a vertex (data cell) to the graph.

- void addVertex (const Vertex &)

  Adds a vertex to the graph.

- void setVertexValue (int, const Data &)

  Sets the Data object to a created vertex.

- void setRMS (Vector< double >)

  Sets the RMS values to the graph vertices.

- void addLinks (const std::string &, const std::string &, double)

  Adds a link between two nodes of the graph.

- void addLinks (int, int, double)

  Adds a link between two nodes of the graph.

- double searchLink (int, int)

  Searches a link between two nodes of the graph.

- Data * searchNode (int, double)

  Returns the data information of a node that is neighbor through a specific link.

- int size ()

  Returns the number of vertices of the graph.

- Vector< int > elements ()

  Calculates the vertices that has a non-empty neighborhood.

- Vector< Data > neighborhood (int)

  Gets the data values of the vertices that are part of the neighborhood of a certain vertex identifier.

- void print ()

  Prints a Graph object.

7.21.2 Public Attributes

- int iDimension
- int * iAccDimensions
- Vector< Vertex > vVertex
- std::map< int, int > mHash
- std::map< std::string, int > mLinks

7.21.3 Detailed Description

Allocates information about a graph that could be a system graph or an application graph.

7.21.4 Constructor & Destructor Documentation

Reconfiguration::Graph::Graph ( )

Constructor of the Graph class.

Returns

*this

Creates an empty object Graph.

Reconfiguration::Graph::Graph ([std::map< std::string, int >]mLinks, int *iDimension,* int * *iAccDimensions*)

Constructor of the Graph class.

Parameters

in	*std::map<std::string,int>*	Linkname<->Offset structure to relate the link string name with the global offset that implies this link.
in	*iDimension*	Problem dimension.
in	*iAccDimensions*	Array of accumulated length per problem dimension. The last position of this array must be equal to the number of vertices of the graph.

Returns

*this

Creates an object Graph and initializes the mLinks map to the values received as argument.

Parameters

in	*mLinks*	Map Linkname<->Offset to relate the link string name with the global offset that implies this link.
in	*iDimension*	Problem dimension.
in	*iAccDimensions*	Array of accumulated length per problem dimension. The last position of this array must be equal to the number of vertices of the graph.

Returns

∗this

Creates an object Graph and initializes the mLinks map to the values received as argument.

Reconfiguration::Graph::Graph ([const Graph &]graph)

Copy constructor of the Graph class.

Parameters

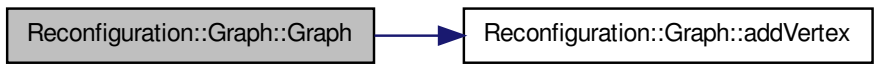
in	*graph*	Graph object to be copied.

Returns

∗this

Creates a new copy of the object Graph.

Here is the call graph for this function:



Reconfiguration::Graph::∼Graph ( ) [virtual]

virtual destructor of the class Graph.

Returns

void

Destroys the Graph object.

7.21.5 Member Function Documentation

void Reconfiguration::Graph::addLinks ([const std::string &]sSource, const std::string & *sDestination,* double *dWeight*)

Adds a link between two nodes of the graph.

Parameters

in	*sSource*	Source vertex name.
in	*sDestination*	Destination vertex name.
in	*dWeight*	Link weigth.

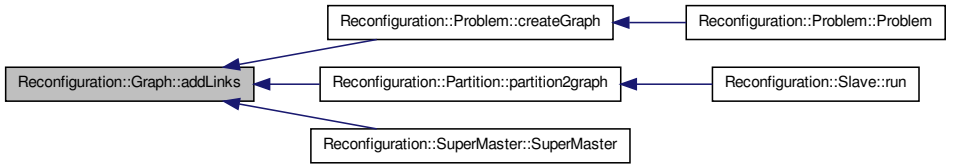
Returns

void

Adds a new link between two nodes of the graph. The names must be valid names (it must exist a vertex with that name).

Here is the call graph for this function:

Here is the caller graph for this function:



void Reconfiguration::Graph::addLinks ([int]iSource, int *iDestination,* double *dWeight*)

Adds a link between two nodes of the graph.

Parameters

in	*iSource*	Source vertex identifier.
in	*iDestination*	Destination vertex identifier.
in	*dWeight*	Link weigth.

Returns

void

Adds a new link between two nodes of the graph. The vertices must exist.

Here is the call graph for this function:

void Reconfiguration::Graph::addVertex ([int]ild, Data *data*)

Adds a vertex (data cell) to the graph.

Parameters

in	*ild*	Vertex identifier.
in	*dWeight*	Data object to inserted.

Returns

void

Adds a specific vertex (Data) to the application graph.

Parameters

in	*ild*	Vertex identifier.
in	*data*	Data object to inserted.

Returns

void

Adds a specific vertex (Data) to the application graph.

Here is the call graph for this function:

void Reconfiguration::Graph::addVertex ([const Vertex &]vertex)

Adds a vertex to the graph.

Parameters

in	*vertex*	Vertex to be added.

Returns

void

Adds a specific vertex to a graph.

void Reconfiguration::Graph::addVertex ([int]iId, double *dCPU,* double *dMemory,* const std::string & *sName*)

Adds a vertex (machine) to the graph.

Parameters

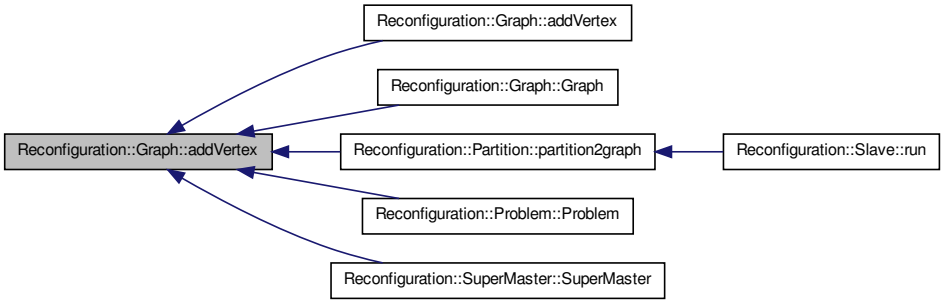
in	*iId*	Vertex identifier.
in	*dCPU*	CPU Frequency.
in	*dMemory*	Free available memory.
in	*sName*	Machine hostname.

Returns

void

Adds a specific vertex (machine) to the system graph.

Here is the caller graph for this function:



Vector< int > Reconfiguration::Graph::elements ( )

Calculates the vertices that has a non-empty neighborhood.

Returns

Vector<int>

Returns a vector of vertex identifiers which neighborhood is not empty.

int Reconfiguration::Graph::idVertex ([const std::string &]sName)

Searches for a certain cluster machine.

Parameters

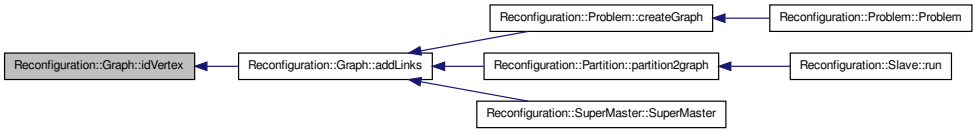
in	*sName*	Hostname of the machine.

Returns

int

Returns the index of the vertex that contains a machine object which name is the specified one or -1 in case of error.

Here is the caller graph for this function:



int Reconfiguration::Graph::idVertex ([int]ild)

Searches for a certain graph vertex.

Parameters

in	*ild*	Vertex identifier.

Returns

int

Returns the index of the vertex which identifier is the one specified.

Vector< Data > Reconfiguration::Graph::neighborhood ([int]id)

Gets the data values of the vertices that are part of the neighborhood of a certain vertex identifier.

Parameters

in	*id*	Vertex identifier.

Returns

Vector<Data>

Returns the vector of Data values of the vertices part of the neighborhood of a certain vertex identifier.

Data Reconfiguration::Graph::operator() ([int]ilndex) const

Access operator ()

Parameters

in	*iIndex*	Vertex index.

Returns

Data object

Returns the Data object allocated at the referred position of the graph (

See also

vVertex).

Data Reconfiguration::Graph::operator() ([int]iIndex, int * *iOffsets*)

Access operator () by distance vector.

Parameters

in	*iIndex*	Vertex index.
in	*iOffsets*	Offsets per dimension.

Returns

Data object

Returns the Data object allocated at a certain position of the graph. The way this vertex is located follows these steps:

Calculates the global offset from the point of view of the actual vertex.

Searches for a link from the vertex pointed by the first argument (v) that has the global offset as weight.

Returns the Data object allocated at that vertex v.

Data & Reconfiguration::Graph::operator() ([int]iIndex)

Access operator ()

Parameters

in	*iIndex*	Vertex index.

Returns

Data&

Returns a reference to the Data object allocated at a specific position of the graph (

See also

vVertex).

Data Reconfiguration::Graph::operator() ([int]iIndex, const std::string & *sName*)

Access operator () by linkname.

Parameters

in	*iIndex*	Vertex index.
in	*sName*	Name of the link that connects to the destination vertex.

Returns

Data object

Returns the Data object allocated at a certain position of the graph. The way this vertex is located follows these steps:

Searches for the link name at mLinks and gets the second field (global Offset).

See also

mLinks

Searches for a link from the vertex pointed by the first argument (v) that has the global offset as weight.

Returns the Data object allocated at that vertex v.

void Reconfiguration::Graph::print ( )

Prints a Graph object.

Returns

void

Prints the properties of the Graph object.

Here is the caller graph for this function:

double Reconfiguration::Graph::searchLink ([int]iSource, int *iDestination*)

Searches a link between two nodes of the graph.

Parameters

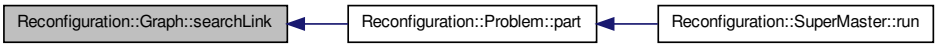
in	*iSource*	Source vertex identifier.
in	*iDestination*	Destination vertex identifier.

Returns

double

Searches a link between two nodes of the graph and returns its weight. The vertices must exist.

Here is the caller graph for this function:



Data * Reconfiguration::Graph::searchNode ([int]iSource, double *dWeight*)

Returns the data information of a node that is neighbor through a specific link.

Parameters

in	*iSource*	Source vertex identifier.
in	*dWeight*	Link weight.

Returns

Data* or NULL in case of error.

Returns a pointer of the data object of a vertex that is part of the neighborhood of the specified vertex and which has a certain link weight.

void Reconfiguration::Graph::setRMS ([Vector< double >]vValues)

Sets the RMS values to the graph vertices.

Parameters

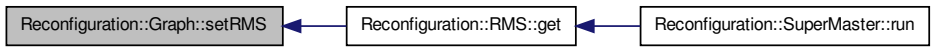
in	*vValues*	Vector of RMS values.

Returns

void

Sets the RMS values to the vertices of the graph. These ones must exist before this function is called. The number of vertices of the graph must be equal to length of the vector received as argument.

Here is the caller graph for this function:



void Reconfiguration::Graph::setVertexValue ([int]iNode, const Data & *dValue*)

Sets the Data object to a created vertex.

Parameters

in	*iNode*	Vertex to be added.
in	*dValue*	Data Object.

Returns

void

Sets the Data information to a vertex that must be created before this function is called.

int Reconfiguration::Graph::size ( )

Returns the number of vertices of the graph.

Returns

int

Returns the length of the vector of vertices.

7.21.6 Member Data Documentation

int * Reconfiguration::Graph::iAccDimensions

iAccDimensions Array of accumulated length per problem dimension. The last position of this array must be equal to the number of vertices of the graph.

int Reconfiguration::Graph::iDimension

iDimension Problem dimension.

std::map< int, int > Reconfiguration::Graph::mHash

Corresponds the vertex identifier with the offset inside the vector, to facilitate following search elements.

std::map< string, int > Reconfiguration::Graph::mLinks

Corresponds the name of the link with the global offset of the link. This global offset depends on the dimensions of the problem.

Vector< Vertex > Reconfiguration::Graph::vVertex

Vector of Vertex object of the graph.

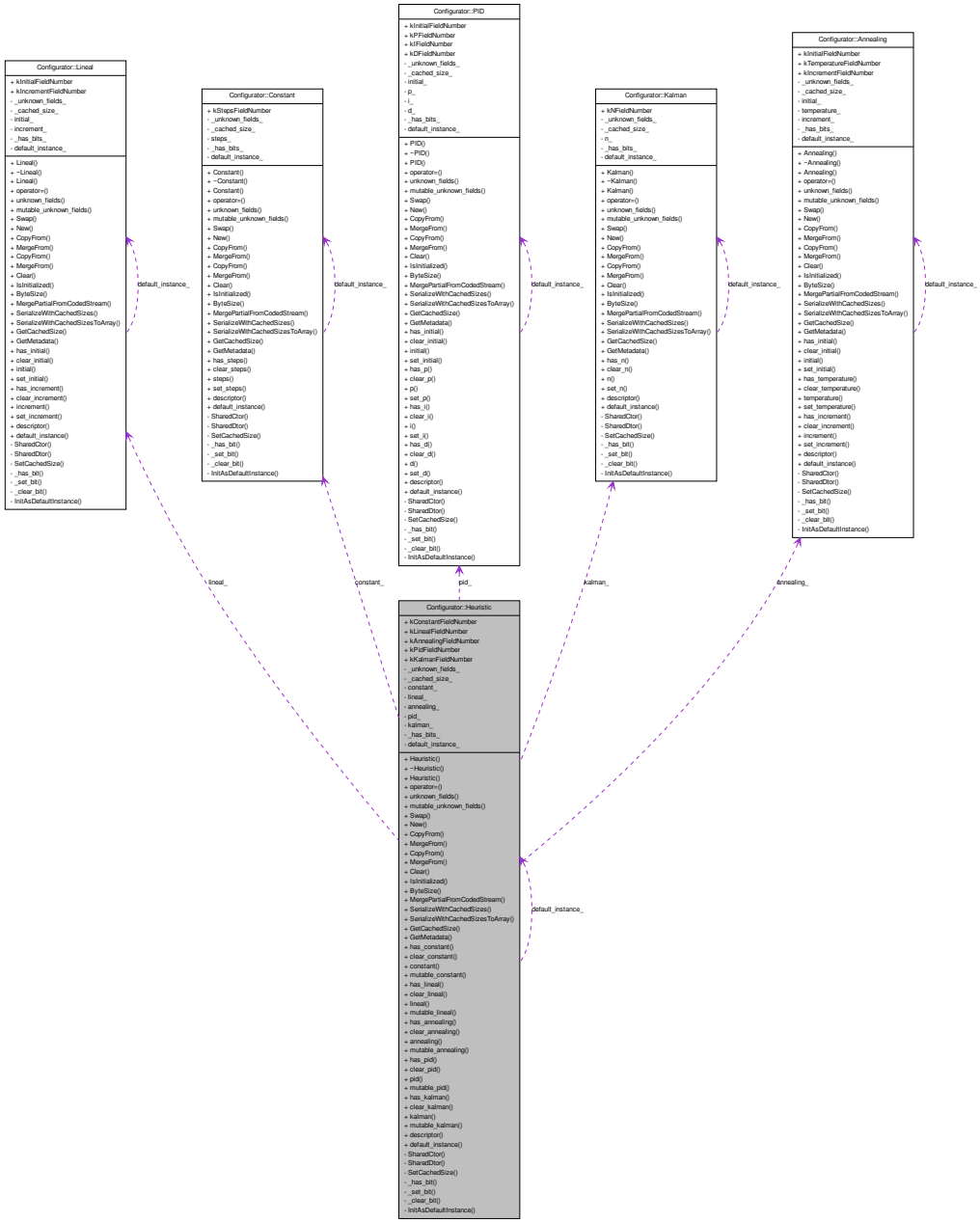
The documentation for this class was generated from the following files:

- Graph.h
- Graph.cpp

7.22 Configurator::Heuristic Class Reference

`#include <confpartitioner.pb.h>`

Collaboration diagram for Configurator::Heuristic:



7.22.1 Public Member Functions

- Heuristic ()

- virtual ~Heuristic ()
- Heuristic (const Heuristic &from)
- Heuristic & operator= (const Heuristic &from)
- const ::google::protobuf::UnknownFieldSet & unknown_fields () const
- inline::google::protobuf::UnknownFieldSet * mutable_unknown_fields ()
- void Swap (Heuristic *other)
- Heuristic * New () const
- void CopyFrom (const ::google::protobuf::Message &from)
- void MergeFrom (const ::google::protobuf::Message &from)
- void CopyFrom (const Heuristic &from)
- void MergeFrom (const Heuristic &from)
- void Clear ()
- bool IsInitialized () const
- int ByteSize () const
- bool MergePartialFromCodedStream (::google::protobuf::io::CodedInputStream *input)
- void SerializeWithCachedSizes (::google::protobuf::io::CodedOutputStream *output) const
- ::google::protobuf::uint8 * SerializeWithCachedSizesToArray (::google::protobuf::uint8 *output) const
- int GetCachedSize () const
- ::google::protobuf::Metadata GetMetadata () const
- bool has_constant () const
- void clear_constant ()
- const ::Configurator::Constant & constant () const
- inline::Configurator::Constant * mutable_constant ()
- bool has_lineal () const
- void clear_lineal ()
- const ::Configurator::Lineal & lineal () const
- inline::Configurator::Lineal * mutable_lineal ()
- bool has_annealing () const
- void clear_annealing ()
- const ::Configurator::Annealing & annealing () const
- inline::Configurator::Annealing * mutable_annealing ()

- bool has_pid () const
- void clear_pid ()
- const ::Configurator::PID & pid () const
- inline::Configurator::PID ∗ mutable_pid ()
- bool has_kalman () const
- void clear_kalman ()
- const ::Configurator::Kalman & kalman () const
- inline::Configurator::Kalman ∗ mutable_kalman ()

7.22.2 Static Public Member Functions

- static const ::google::protobuf::Descriptor ∗ descriptor ()
- static const Heuristic & default_instance ()

7.22.3 Static Public Attributes

- static const int kConstantFieldNumber = 1
- static const int kLinealFieldNumber = 2
- static const int kAnnealingFieldNumber = 3
- static const int kPidFieldNumber = 4
- static const int kKalmanFieldNumber = 5

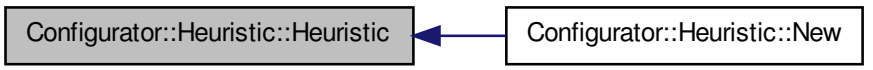
7.22.4 Friends

- void protobuf_AddDesc_confpartitioner_2eproto ()
- void protobuf_AssignDesc_confpartitioner_2eproto ()
- void protobuf_ShutdownFile_confpartitioner_2eproto ()

7.22.5 Constructor & Destructor Documentation

Configurator::Heuristic::Heuristic ( )

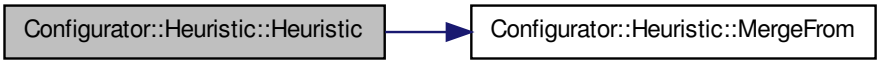
Here is the caller graph for this function:



Configurator::Heuristic::~Heuristic ( ) [virtual]

Configurator::Heuristic::Heuristic ([const Heuristic &]from)

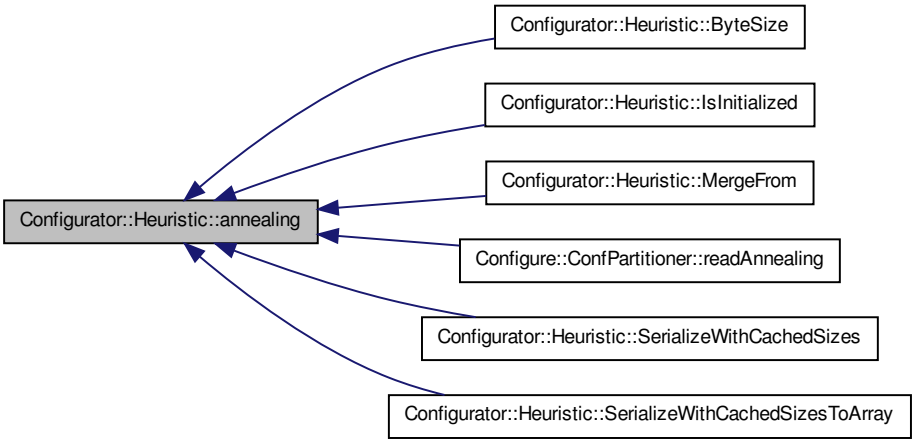
Here is the call graph for this function:



7.22.6 Member Function Documentation

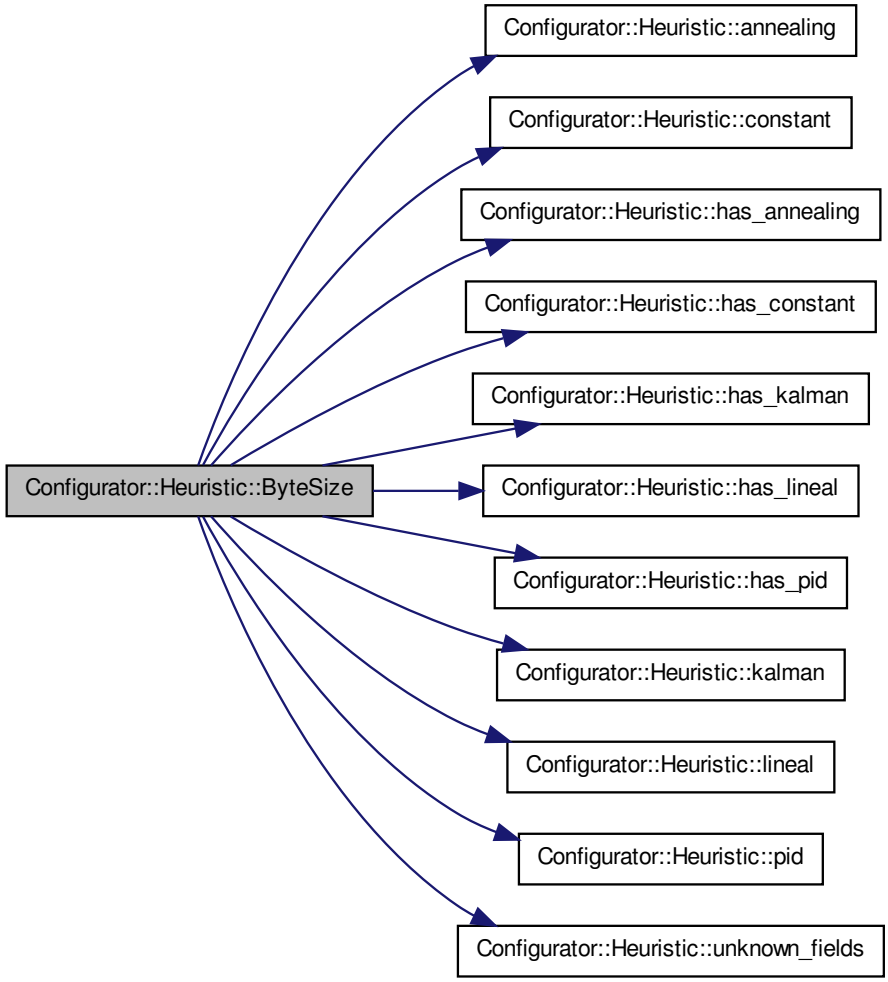
const ::Configurator::Annealing & Configurator::Heuristic::annealing ( ) const [inline]

Here is the caller graph for this function:



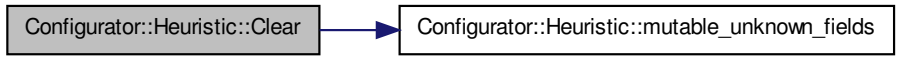
int Configurator::Heuristic::ByteSize ( ) const

Here is the call graph for this function:

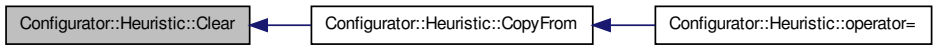


void Configurator::Heuristic::Clear ( )

Here is the call graph for this function:



Here is the caller graph for this function:



void Configurator::Heuristic::clear_annealing ( ) [inline]

void Configurator::Heuristic::clear_constant ( ) [inline]

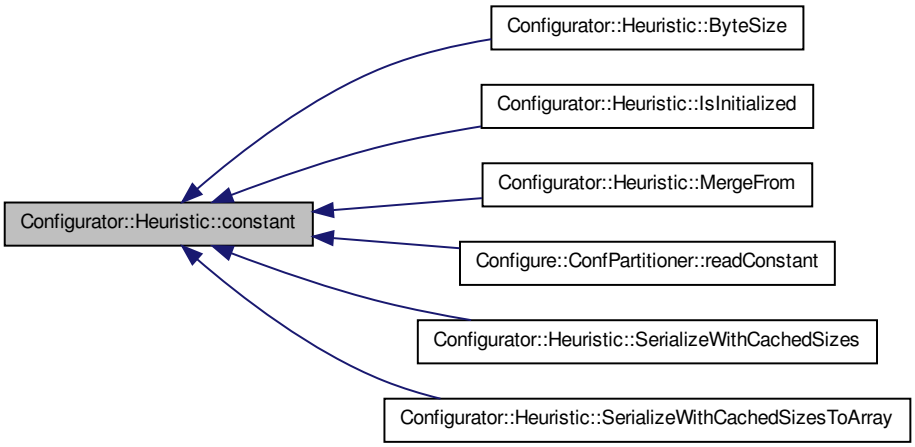
void Configurator::Heuristic::clear_kalman ( ) [inline]

void Configurator::Heuristic::clear_lineal ( ) [inline]

void Configurator::Heuristic::clear_pid ( ) [inline]

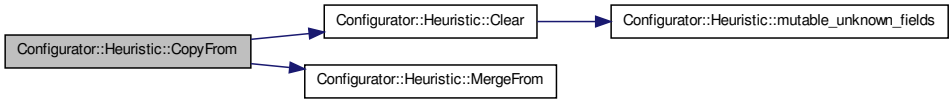
const ::Configurator::Constant & Configurator::Heuristic::constant ( ) const [inline]

Here is the caller graph for this function:



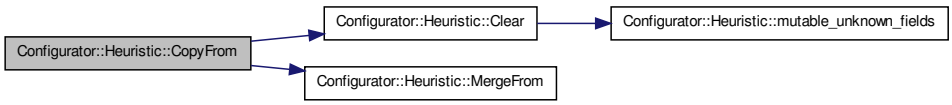
void Configurator::Heuristic::CopyFrom ([const Heuristic &]from)

Here is the call graph for this function:



void Configurator::Heuristic::CopyFrom ([const ::google::protobuf::Message &]from)

Here is the call graph for this function:



Here is the caller graph for this function:

const Heuristic & Configurator::Heuristic::default_instance ( ) [static]

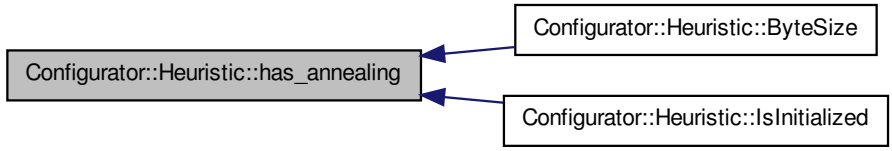
const ::google::protobuf::Descriptor * Configurator::Heuristic::descriptor ( ) [static]

int Configurator::Heuristic::GetCachedSize ( ) const [inline]

google::protobuf::Metadata Configurator::Heuristic::GetMetadata ( ) const

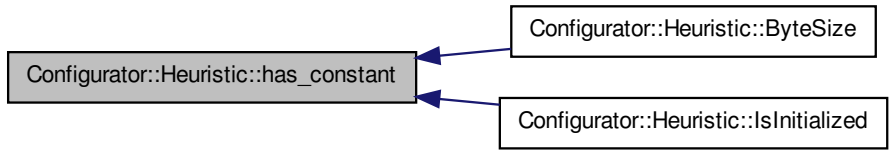
bool Configurator::Heuristic::has_annealing ( ) const [inline]

Here is the caller graph for this function:



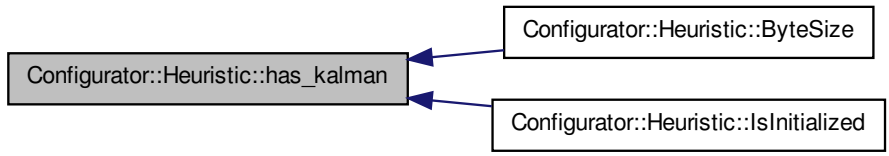
bool Configurator::Heuristic::has_constant ( ) const [inline]

Here is the caller graph for this function:



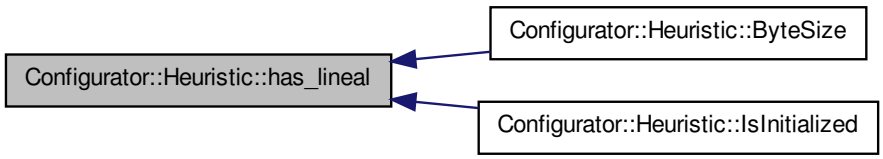
bool Configurator::Heuristic::has_kalman ( ) const [inline]

Here is the caller graph for this function:



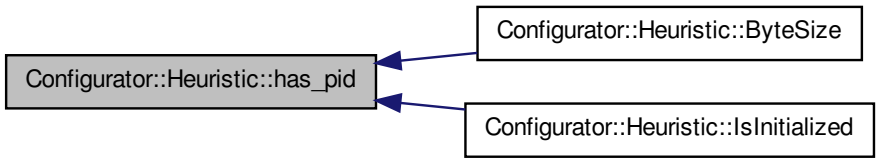
bool Configurator::Heuristic::has_lineal ( ) const [inline]

Here is the caller graph for this function:



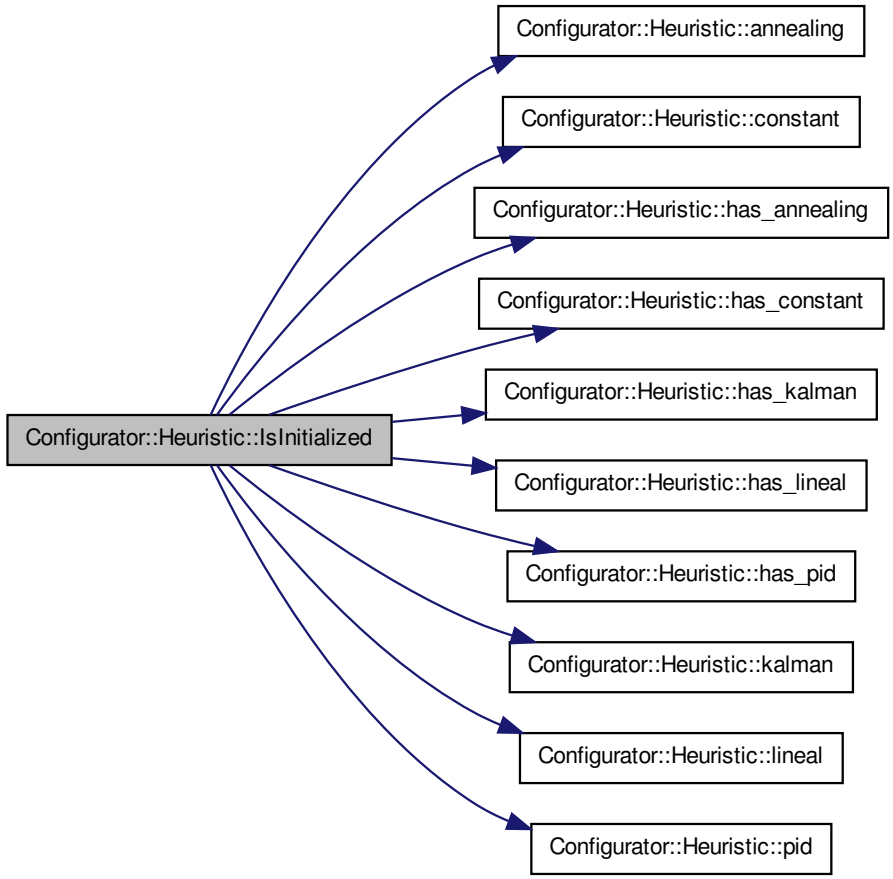
bool Configurator::Heuristic::has_pid ( ) const [inline]

Here is the caller graph for this function:



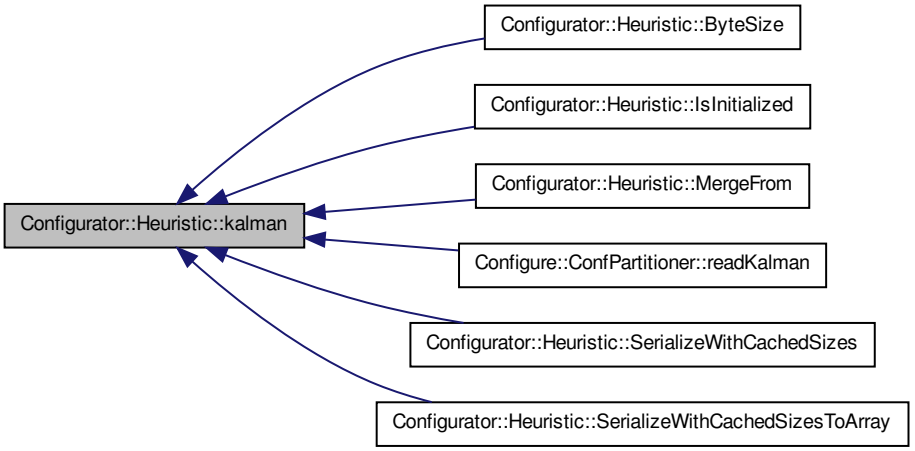
bool Configurator::Heuristic::IsInitialized ( ) const

Here is the call graph for this function:



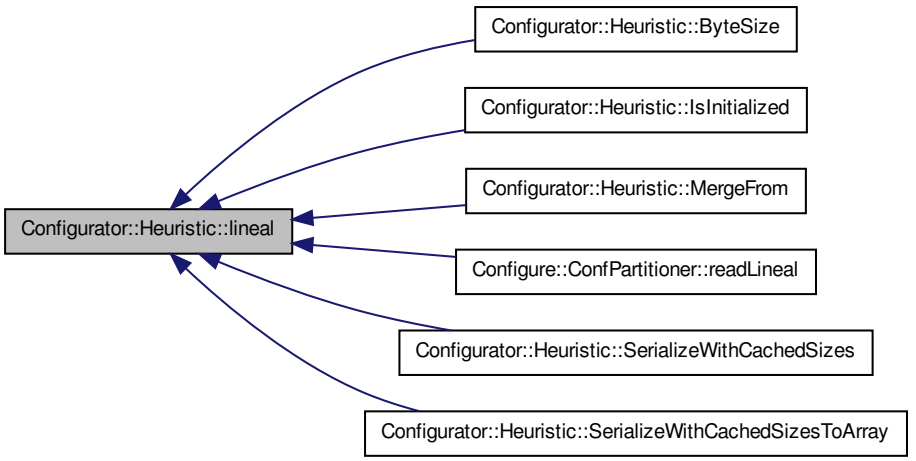
const ::Configurator::Kalman & Configurator::Heuristic::kalman ( ) const [inline]

Here is the caller graph for this function:



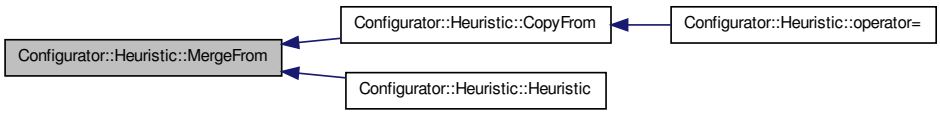
const ::Configurator::Lineal & Configurator::Heuristic::lineal ( ) const `[inline]`

Here is the caller graph for this function:



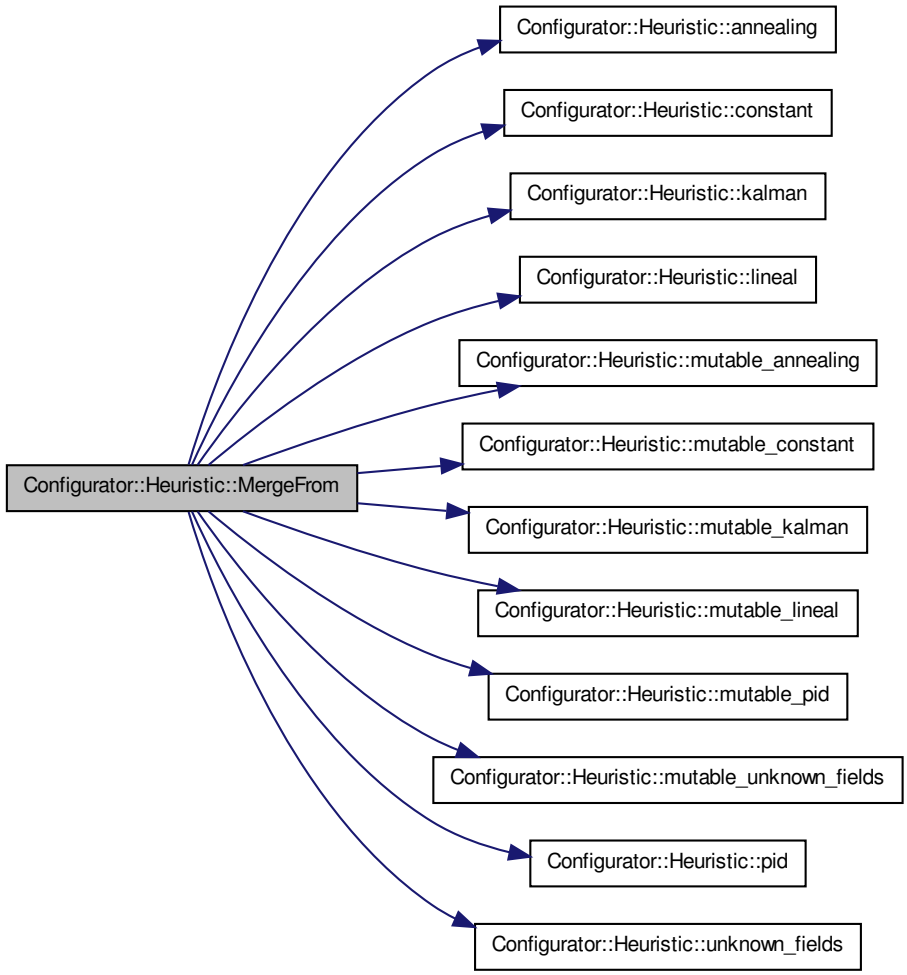
void Configurator::Heuristic::MergeFrom ([const ::google::protobuf::Message &]from)

Here is the caller graph for this function:



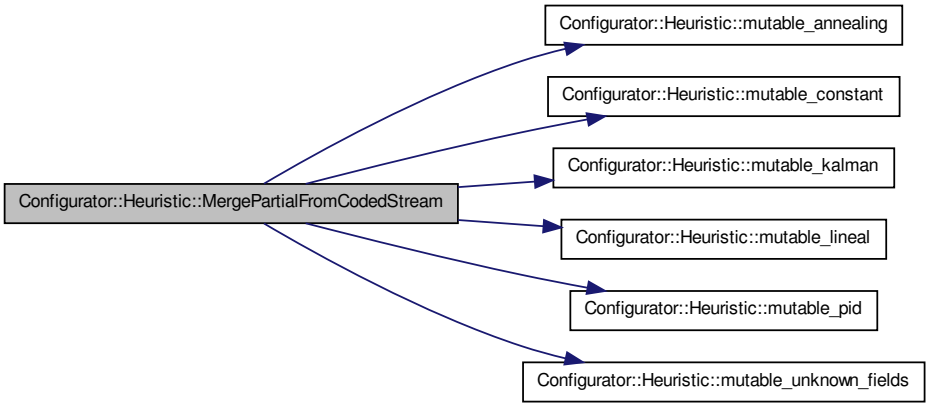
void Configurator::Heuristic::MergeFrom ([const Heuristic &]from)

Here is the call graph for this function:



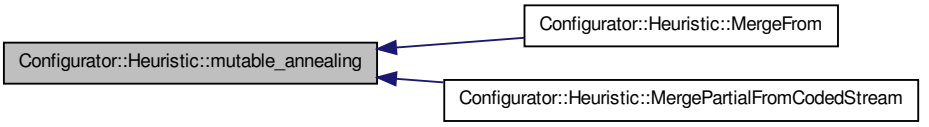
bool Configurator::Heuristic::MergePartialFromCodedStream ([::google::protobuf::io::CodedInputStream *]input)

Here is the call graph for this function:



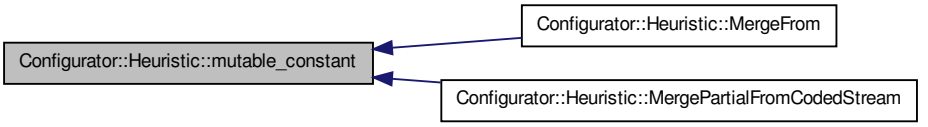
Configurator::Annealing * Configurator::Heuristic::mutable_annealing ( ) [inline]

Here is the caller graph for this function:



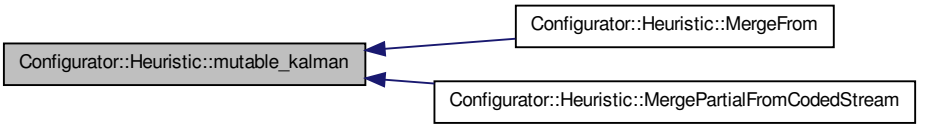
Configurator::Constant * Configurator::Heuristic::mutable_constant ( ) [inline]

Here is the caller graph for this function:



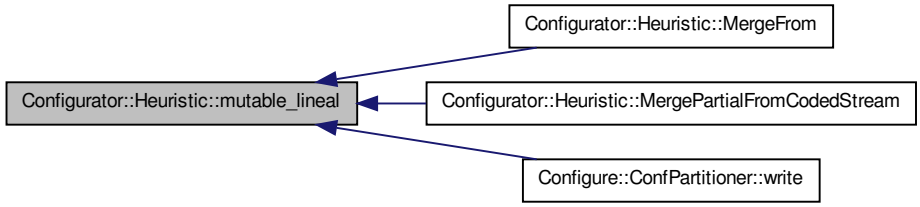
Configurator::Kalman * Configurator::Heuristic::mutable_kalman ( ) [inline]

Here is the caller graph for this function:



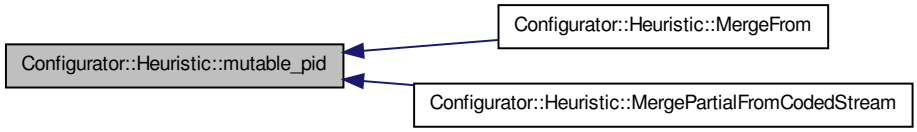
Configurator::Lineal * Configurator::Heuristic::mutable_lineal ( ) [inline]

Here is the caller graph for this function:



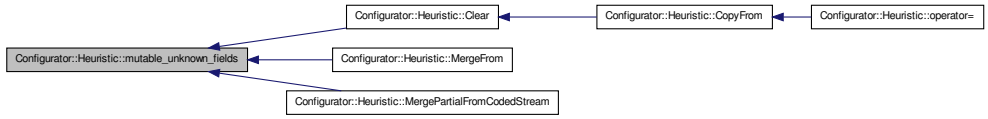
Configurator::PID * Configurator::Heuristic::mutable_pid ( ) [inline]

Here is the caller graph for this function:



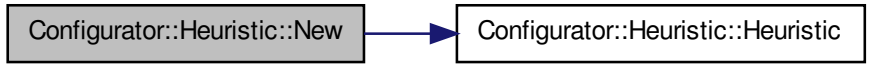
inline ::google::protobuf::UnknownFieldSet* Configurator::Heuristic::mutable_-unknown_fields ( ) [inline]

Here is the caller graph for this function:



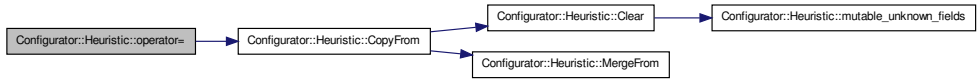
Heuristic * Configurator::Heuristic::New ( ) const

Here is the call graph for this function:



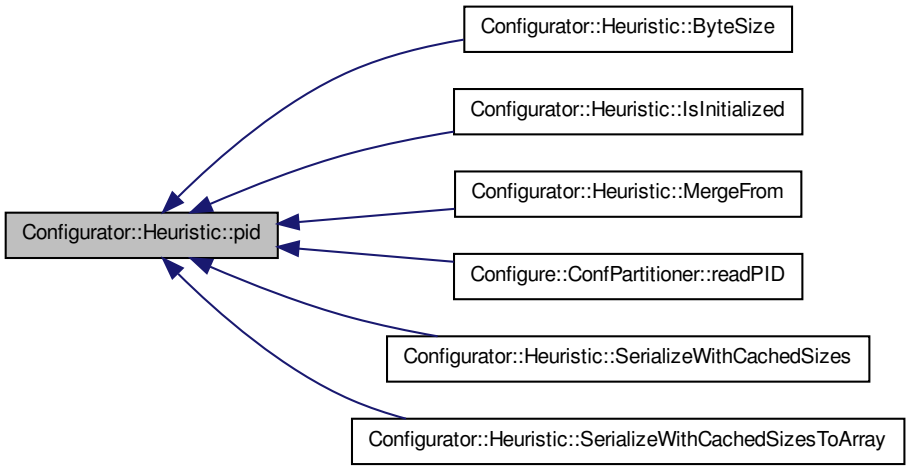
Heuristic& Configurator::Heuristic::operator= ([const Heuristic &]from) [inline]

Here is the call graph for this function:



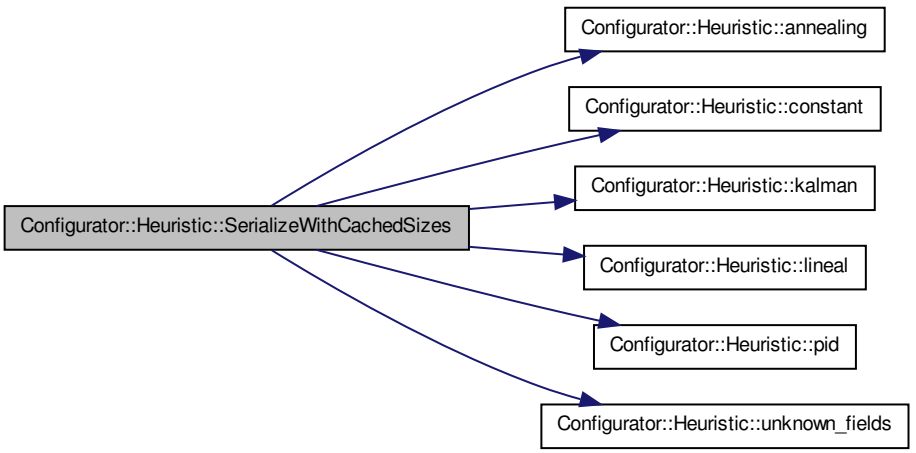
const ::Configurator::PID & Configurator::Heuristic::pid ( ) const [inline]

Here is the caller graph for this function:



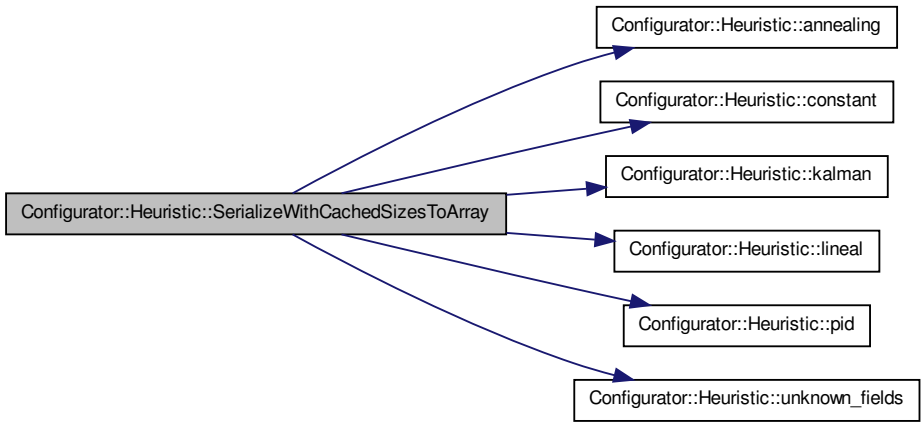
void Configurator::Heuristic::SerializeWithCachedSizes ([::google::protobuf::io::CodedOutputStream ∗]output) const

Here is the call graph for this function:



google::protobuf::uint8 ∗ Configurator::Heuristic::SerializeWithCachedSizesToArray ([::google::protobuf::uint8 ∗]output) const

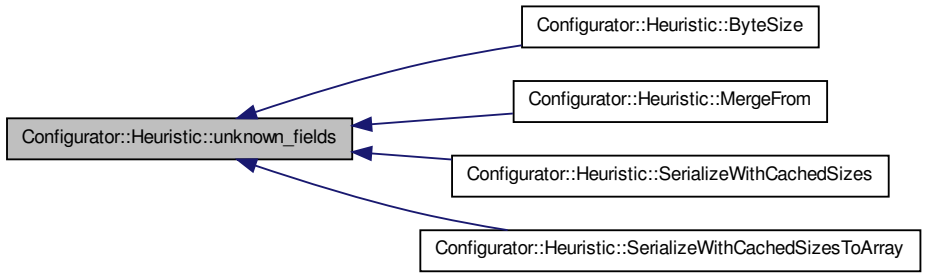
Here is the call graph for this function:



void Configurator::Heuristic::Swap ([Heuristic *]other)

const ::google::protobuf::UnknownFieldSet& Configurator::Heuristic::unknown_-fields ( ) const [inline]

Here is the caller graph for this function:



7.22.7 Friends And Related Function Documentation

void protobuf_AddDesc_confpartitioner_2eproto ( ) [friend]

void protobuf_AssignDesc_confpartitioner_2eproto ( ) [friend]

void protobuf_ShutdownFile_confpartitioner_2eproto ( ) [friend]

7.22.8 Member Data Documentation

const int Configurator::Heuristic::kAnnealingFieldNumber = 3 `[static]`

const int Configurator::Heuristic::kConstantFieldNumber = 1 `[static]`

const int Configurator::Heuristic::kKalmanFieldNumber = 5 `[static]`

const int Configurator::Heuristic::kLinealFieldNumber = 2 `[static]`

const int Configurator::Heuristic::kPidFieldNumber = 4 `[static]`

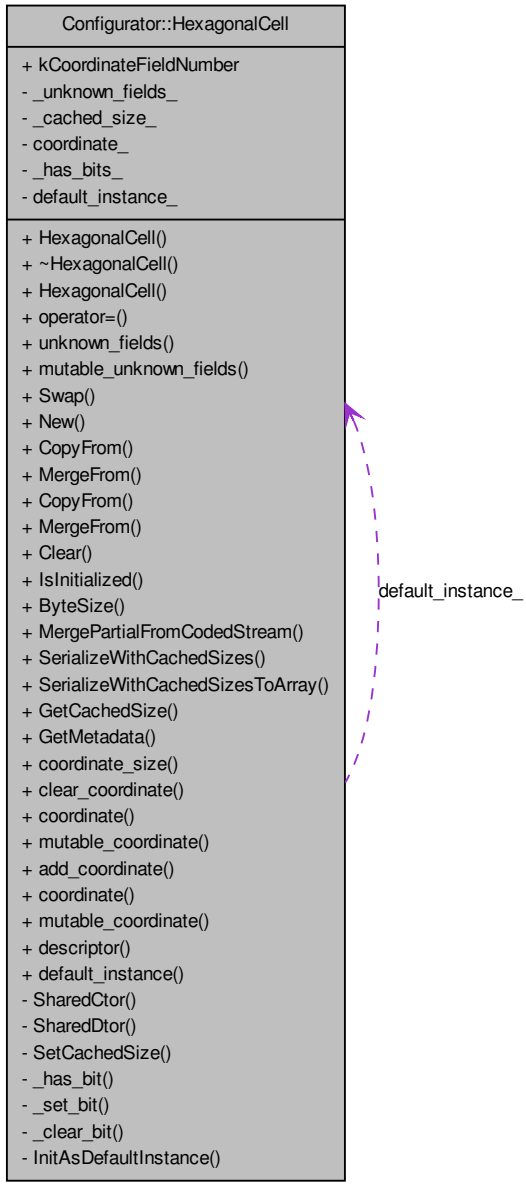
The documentation for this class was generated from the following files:

- confpartitioner.pb.h
- confpartitioner.pb.cc

7.23 Configurator::HexagonalCell Class Reference

`#include <confproblem.pb.h>`

Collaboration diagram for Configurator::HexagonalCell:



7.23.1 Public Member Functions

- HexagonalCell ()
- virtual ∼HexagonalCell ()
- HexagonalCell (const HexagonalCell &from)
- HexagonalCell & operator= (const HexagonalCell &from)
- const ::google::protobuf::UnknownFieldSet & unknown_fields () const
- inline::google::protobuf::UnknownFieldSet ∗ mutable_unknown_fields ()
- void Swap (HexagonalCell ∗other)
- HexagonalCell ∗ New () const
- void CopyFrom (const ::google::protobuf::Message &from)
- void MergeFrom (const ::google::protobuf::Message &from)
- void CopyFrom (const HexagonalCell &from)
- void MergeFrom (const HexagonalCell &from)
- void Clear ()
- bool IsInitialized () const
- int ByteSize () const
- bool MergePartialFromCodedStream (::google::protobuf::io::CodedInputStream ∗input)
- void SerializeWithCachedSizes (::google::protobuf::io::CodedOutputStream ∗output) const
- ::google::protobuf::uint8 ∗ SerializeWithCachedSizesToArray (::google::protobuf::uint8 ∗output) const
- int GetCachedSize () const
- ::google::protobuf::Metadata GetMetadata () const
- int coordinate_size () const
- void clear_coordinate ()
- const ::Configurator::Coordinate & coordinate (int index) const
- inline::Configurator::Coordinate ∗ mutable_coordinate (int index)
- inline::Configurator::Coordinate ∗ add_coordinate ()
- const ::google::protobuf::RepeatedPtrField< ::Configurator::Coordinate > & coordinate () const
- inline::google::protobuf::RepeatedPtrField< ::Configurator::Coordinate > ∗ mutable_-coordinate ()

7.23.2 Static Public Member Functions

- static const ::google::protobuf::Descriptor * descriptor ()
- static const HexagonalCell & default_instance ()

7.23.3 Static Public Attributes

- static const int kCoordinateFieldNumber = 1

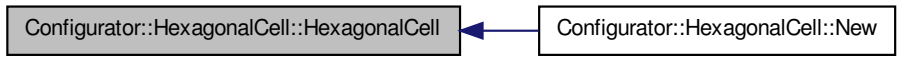
7.23.4 Friends

- void protobuf_AddDesc_confproblem_2eproto ()
- void protobuf_AssignDesc_confproblem_2eproto ()
- void protobuf_ShutdownFile_confproblem_2eproto ()

7.23.5 Constructor & Destructor Documentation

Configurator::HexagonalCell::HexagonalCell ( )

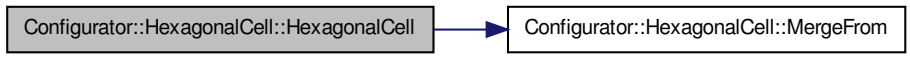
Here is the caller graph for this function:



Configurator::HexagonalCell::~HexagonalCell ( ) `[virtual]`

Configurator::HexagonalCell::HexagonalCell ([const HexagonalCell &]from)

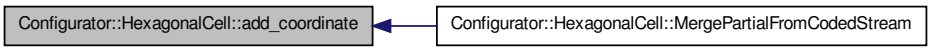
Here is the call graph for this function:



7.23.6 Member Function Documentation

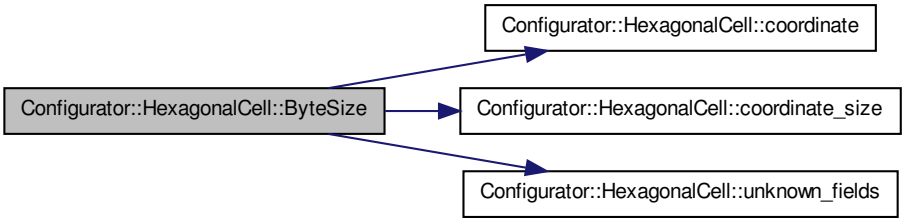
Configurator::Coordinate * Configurator::HexagonalCell::add_coordinate ( ) `[inline]`

Here is the caller graph for this function:



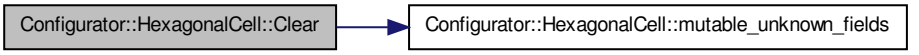
int Configurator::HexagonalCell::ByteSize ( ) const

Here is the call graph for this function:

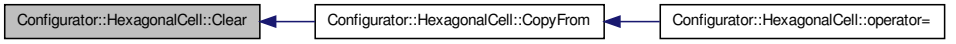


void Configurator::HexagonalCell::Clear ( )

Here is the call graph for this function:



Here is the caller graph for this function:

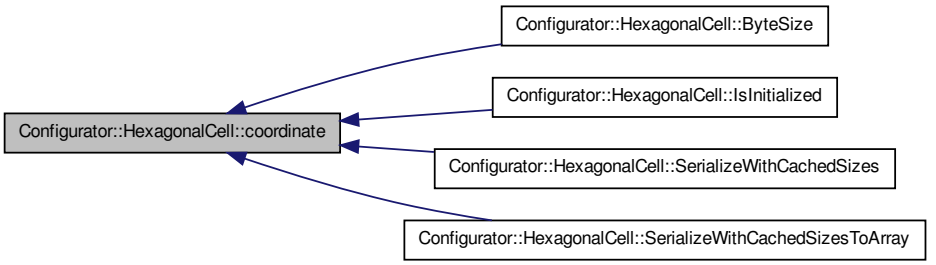


void Configurator::HexagonalCell::clear_coordinate ( ) [inline]

const ::Configurator::Coordinate & Configurator::HexagonalCell::coordinate ([int]index) const [inline]

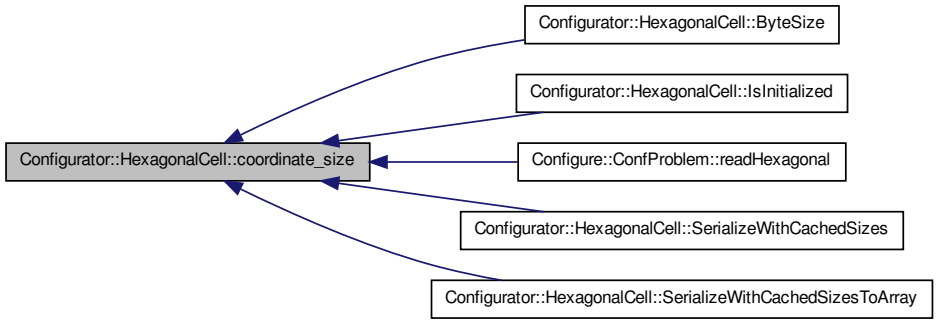
const ::google::protobuf::RepeatedPtrField<::Configurator::Coordinate > & Configurator::HexagonalCell::coordinate ( ) const [inline]

Here is the caller graph for this function:



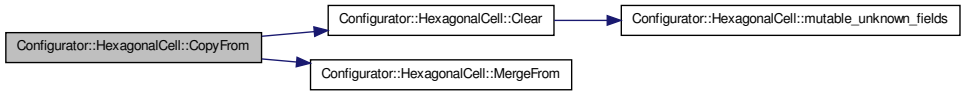
int Configurator::HexagonalCell::coordinate_size ( ) const [inline]

Here is the caller graph for this function:



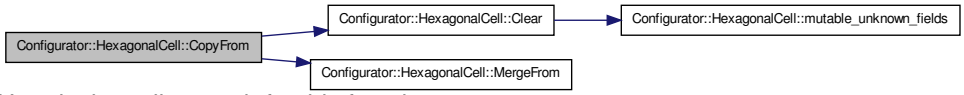
void Configurator::HexagonalCell::CopyFrom ([const HexagonalCell &]from)

Here is the call graph for this function:



void Configurator::HexagonalCell::CopyFrom ([const ::google::protobuf::Message &]from)

Here is the call graph for this function:



Here is the caller graph for this function:

const HexagonalCell & Configurator::HexagonalCell::default_instance ( ) [static]

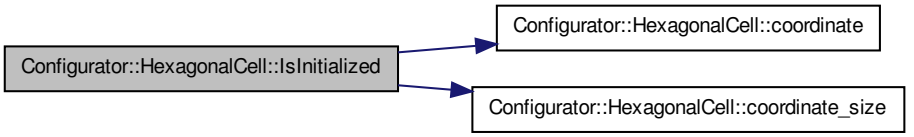
const ::google::protobuf::Descriptor * Configurator::HexagonalCell::descriptor ( ) [static]

int Configurator::HexagonalCell::GetCachedSize ( ) const [inline]

google::protobuf::Metadata Configurator::HexagonalCell::GetMetadata ( ) const

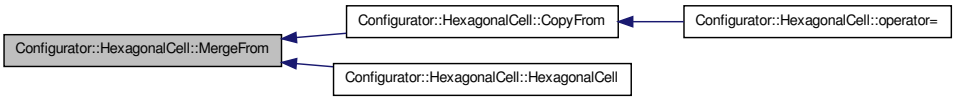
bool Configurator::HexagonalCell::IsInitialized ( ) const

Here is the call graph for this function:



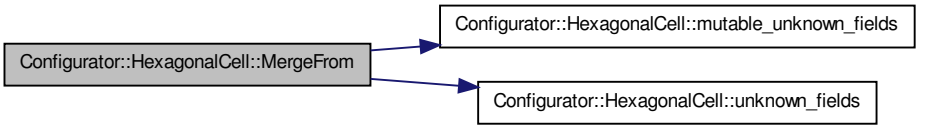
void Configurator::HexagonalCell::MergeFrom ([const ::google::protobuf::Message &]from)

Here is the caller graph for this function:



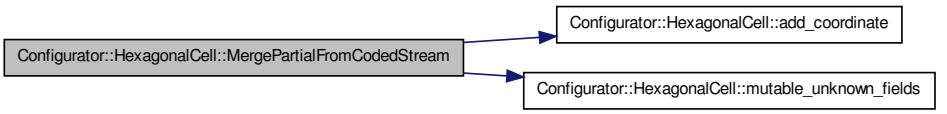
void Configurator::HexagonalCell::MergeFrom ([const HexagonalCell &]from)

Here is the call graph for this function:



bool Configurator::HexagonalCell::MergePartialFromCodedStream ([::google::protobuf::io::CodedInputStream ∗]input)

Here is the call graph for this function:

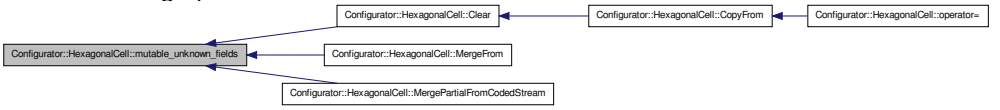


Configurator::Coordinate ∗ Configurator::HexagonalCell::mutable_coordinate ([int]index) [inline]

google::protobuf::RepeatedPtrField<::Configurator::Coordinate > ∗ Configurator::HexagonalCell::mutable_coordinate ( ) [inline]

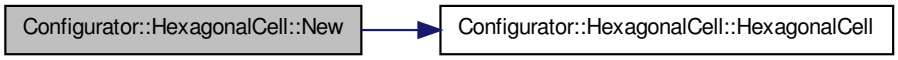
inline ::google::protobuf::UnknownFieldSet∗ Configurator::HexagonalCell::mutable_-unknown_fields ( ) [inline]

Here is the caller graph for this function:



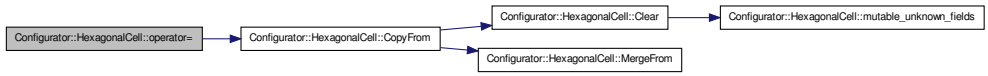
HexagonalCell * Configurator::HexagonalCell::New ( ) const

Here is the call graph for this function:



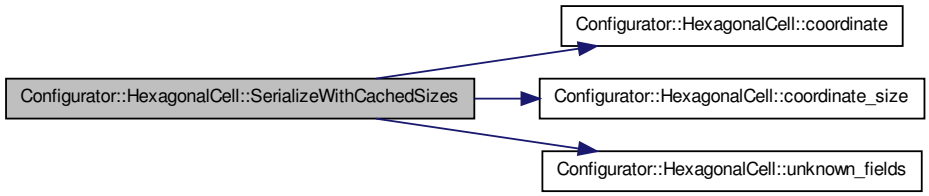
HexagonalCell& Configurator::HexagonalCell::operator= ([const HexagonalCell &]from) [inline]

Here is the call graph for this function:



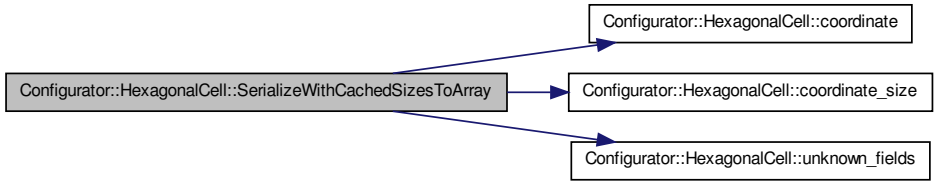
void Configurator::HexagonalCell::SerializeWithCachedSizes ([::google::protobuf::io::CodedOutputStream *]output) const

Here is the call graph for this function:



google::protobuf::uint8 * Configurator::HexagonalCell::SerializeWithCachedSizesToArray ([::google::protobuf::uint8 *]output) const

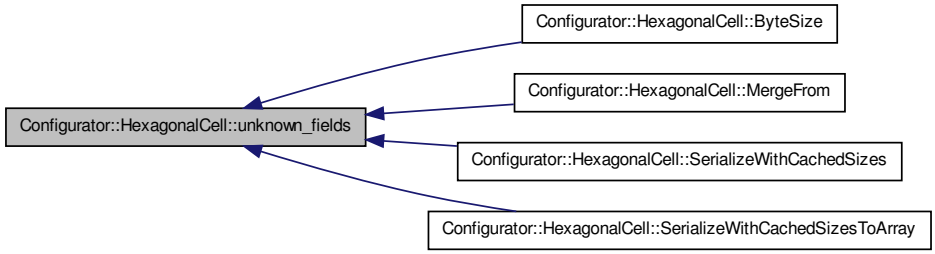
Here is the call graph for this function:



void Configurator::HexagonalCell::Swap ([HexagonalCell *]other)

const ::google::protobuf::UnknownFieldSet& Configurator::HexagonalCell::unknown_-fields ( ) const [inline]

Here is the caller graph for this function:



7.23.7 Friends And Related Function Documentation

void protobuf_AddDesc_confproblem_2eproto ( ) [friend]

void protobuf_AssignDesc_confproblem_2eproto ( ) [friend]

void protobuf_ShutdownFile_confproblem_2eproto ( ) [friend]

7.23.8 Member Data Documentation

const int Configurator::HexagonalCell::kCoordinateFieldNumber = 1 [static]

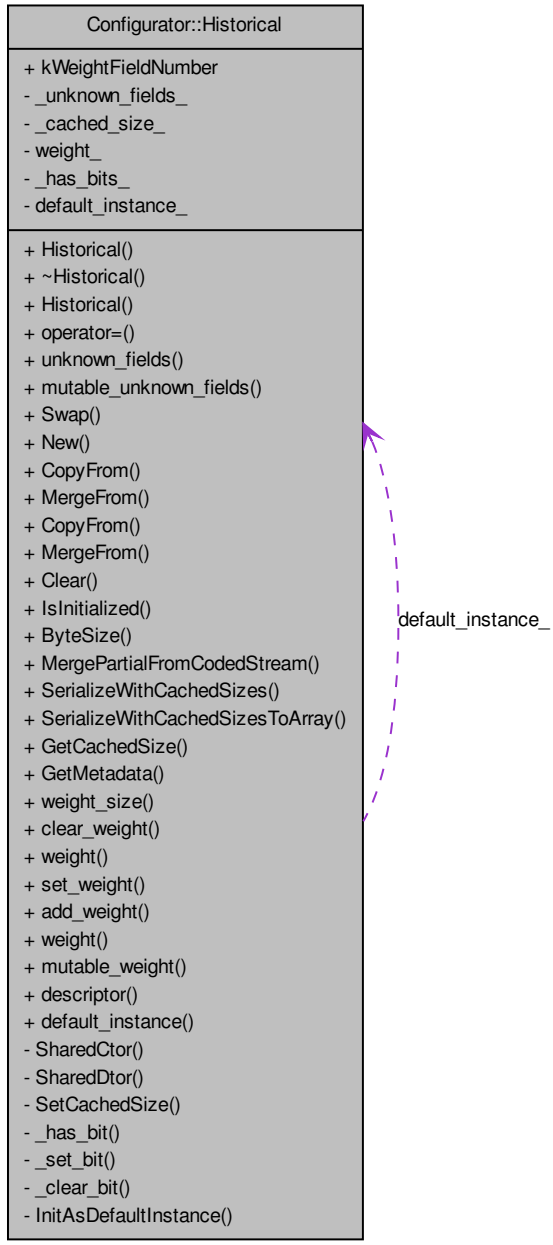
The documentation for this class was generated from the following files:

- confproblem.pb.h
- confproblem.pb.cc

7.24 Configurator::Historical Class Reference

`#include <confloadbalancer.pb.h>`

Collaboration diagram for Configurator::Historical:



7.24.1 Public Member Functions

- Historical ()
- virtual ∼Historical ()
- Historical (const Historical &from)
- Historical & operator= (const Historical &from)
- const ::google::protobuf::UnknownFieldSet & unknown_fields () const
- inline::google::protobuf::UnknownFieldSet ∗ mutable_unknown_fields ()
- void Swap (Historical ∗other)
- Historical ∗ New () const
- void CopyFrom (const ::google::protobuf::Message &from)
- void MergeFrom (const ::google::protobuf::Message &from)
- void CopyFrom (const Historical &from)
- void MergeFrom (const Historical &from)
- void Clear ()
- bool IsInitialized () const
- int ByteSize () const
- bool MergePartialFromCodedStream (::google::protobuf::io::CodedInputStream ∗input)
- void SerializeWithCachedSizes (::google::protobuf::io::CodedOutputStream ∗output) const
- ::google::protobuf::uint8 ∗ SerializeWithCachedSizesToArray (::google::protobuf::uint8 ∗output) const
- int GetCachedSize () const
- ::google::protobuf::Metadata GetMetadata () const
- int weight_size () const
- void clear_weight ()
- double weight (int index) const
- void set_weight (int index, double value)
- void add_weight (double value)
- const ::google::protobuf::RepeatedField< double > & weight () const
- inline::google::protobuf::RepeatedField< double > ∗ mutable_weight ()

7.24.2 Static Public Member Functions

- static const ::google::protobuf::Descriptor * descriptor ()
- static const Historical & default_instance ()

7.24.3 Static Public Attributes

- static const int kWeightFieldNumber = 1

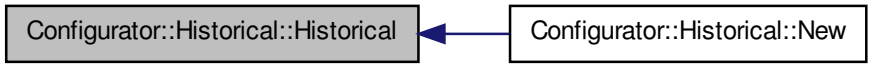
7.24.4 Friends

- void protobuf_AddDesc_confloadbalancer_2eproto ()
- void protobuf_AssignDesc_confloadbalancer_2eproto ()
- void protobuf_ShutdownFile_confloadbalancer_2eproto ()

7.24.5 Constructor & Destructor Documentation

Configurator::Historical::Historical ( )

Here is the caller graph for this function:



Configurator::Historical::~Historical ( ) [virtual]

Configurator::Historical::Historical ([const Historical &]from)

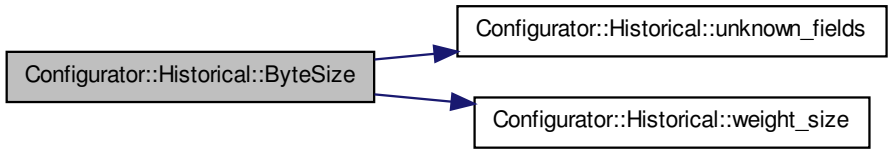
Here is the call graph for this function:

7.24.6 Member Function Documentation

void Configurator::Historical::add_weight ([double]value) [inline]

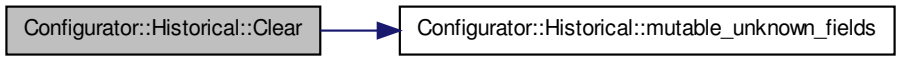
int Configurator::Historical::ByteSize ( ) const

Here is the call graph for this function:

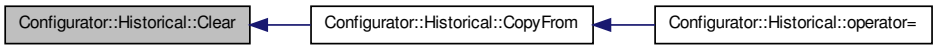


void Configurator::Historical::Clear ( )

Here is the call graph for this function:



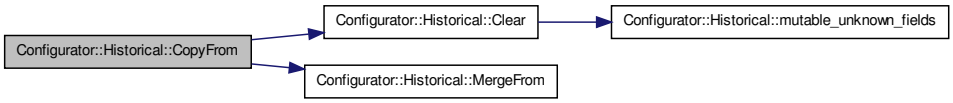
Here is the caller graph for this function:



void Configurator::Historical::clear_weight ( ) `[inline]`

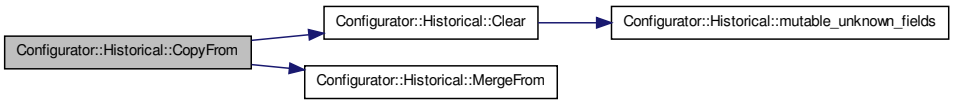
void Configurator::Historical::CopyFrom ([const Historical &]from)

Here is the call graph for this function:



void Configurator::Historical::CopyFrom ([const ::google::protobuf::Message &]from)

Here is the call graph for this function:



Here is the caller graph for this function:

const Historical & Configurator::Historical::default_instance ( ) [static]

const ::google::protobuf::Descriptor * Configurator::Historical::descriptor ( ) [static]

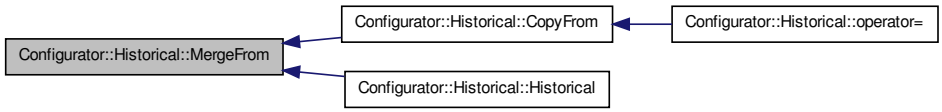
int Configurator::Historical::GetCachedSize ( ) const [inline]

google::protobuf::Metadata Configurator::Historical::GetMetadata ( ) const

bool Configurator::Historical::IsInitialized ( ) const

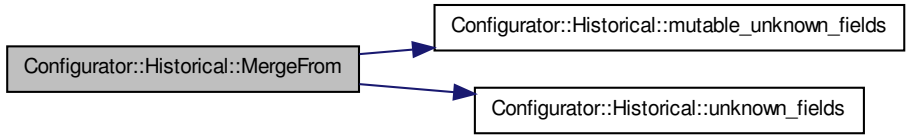
void Configurator::Historical::MergeFrom ([const ::google::protobuf::Message &]from)

Here is the caller graph for this function:



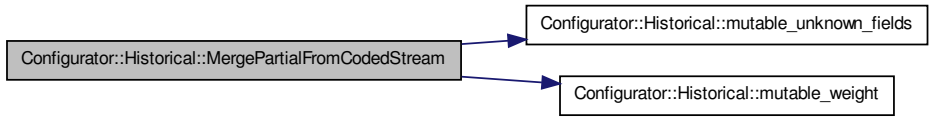
void Configurator::Historical::MergeFrom ([const Historical &]from)

Here is the call graph for this function:



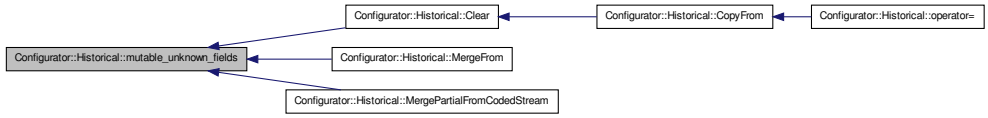
bool Configurator::Historical::MergePartialFromCodedStream ([::google::protobuf::io::CodedInputStream *]input)

Here is the call graph for this function:



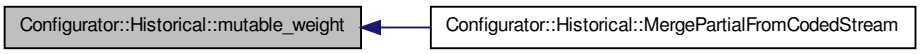
inline ::google::protobuf::UnknownFieldSet* Configurator::Historical::mutable_-unknown_fields ( ) [inline]

Here is the caller graph for this function:



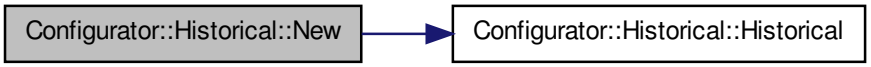
google::protobuf::RepeatedField< double > * Configurator::Historical::mutable_weight ( ) [inline]

Here is the caller graph for this function:



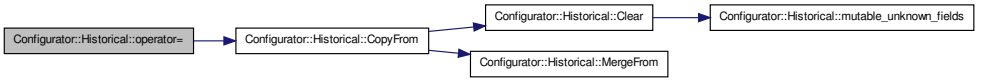
Historical * Configurator::Historical::New ( ) const

Here is the call graph for this function:



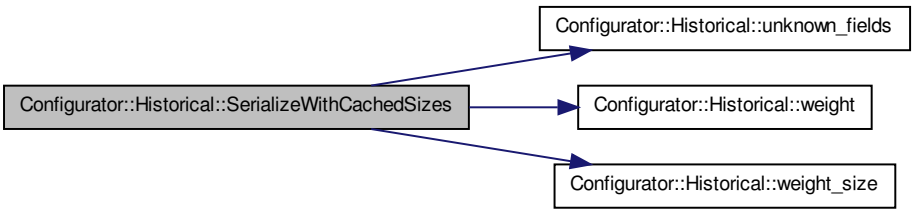
Historical& Configurator::Historical::operator= ([const Historical &]from) [inline]

Here is the call graph for this function:



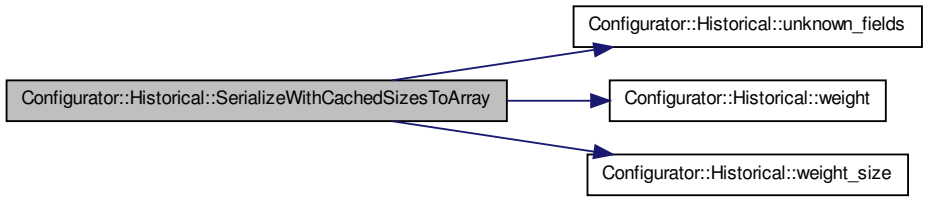
void Configurator::Historical::SerializeWithCachedSizes ([::google::protobuf::io::CodedOutputStream *]output) const

Here is the call graph for this function:



google::protobuf::uint8 * Configurator::Historical::SerializeWithCachedSizesToArray ([::google::protobuf::uint8 *]output) const

Here is the call graph for this function:

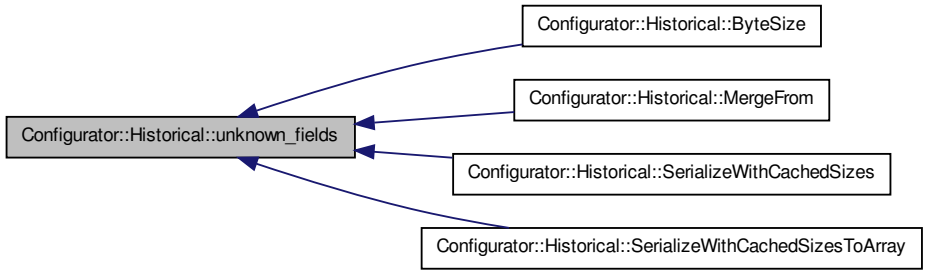


void Configurator::Historical::set_weight ([int]index, double *value*) `[inline]`

void Configurator::Historical::Swap ([Historical ∗]other)

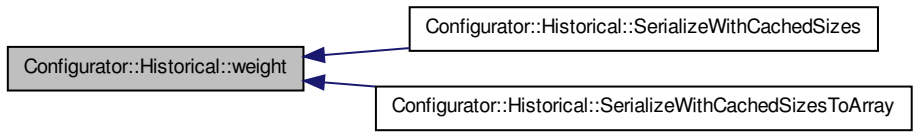
const ::google::protobuf::UnknownFieldSet& Configurator::Historical::unknown_-fields ( ) const `[inline]`

Here is the caller graph for this function:



const ::google::protobuf::RepeatedField< double > & Configurator::Historical::weight ( ) const `[inline]`

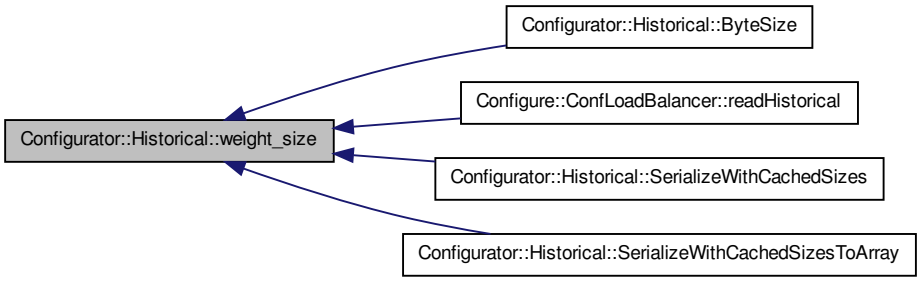
Here is the caller graph for this function:



double Configurator::Historical::weight ([int]index) const `[inline]`

int Configurator::Historical::weight_size ( ) const `[inline]`

Here is the caller graph for this function:



7.24.7 Friends And Related Function Documentation

void protobuf_AddDesc_confloadbalancer_2eproto ( ) [friend]

void protobuf_AssignDesc_confloadbalancer_2eproto ( ) [friend]

void protobuf_ShutdownFile_confloadbalancer_2eproto ( ) [friend]

7.24.8 Member Data Documentation

const int Configurator::Historical::kWeightFieldNumber = 1 [static]

The documentation for this class was generated from the following files:

- confloadbalancer.pb.h
- confloadbalancer.pb.cc

7.25 ID Class Reference

`#include <Min.h>`

7.25.1 Detailed Description

Needed to sort the elements of the queue for the IGrouping interface.

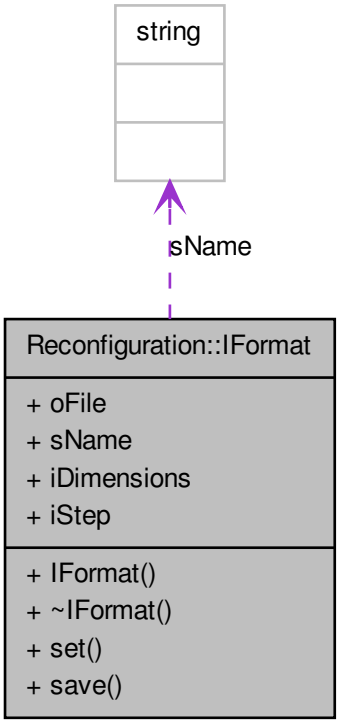
The documentation for this class was generated from the following file:

- Min.h

7.26 Reconfiguration::IFormat Class Reference

`#include <IFormat.h>`

Collaboration diagram for Reconfiguration::IFormat:



7.26.1 Public Member Functions

- IFormat ()

  Constructor of the IFormat class.

- virtual ∼IFormat ()

  virtual destructor of the class IFormat.

- void set (std::string, int, int ∗)

  Sets the properties of the object.

- virtual void save (std::stringstream ∗)=0

  Saves the user information into a file.

7.26.2 Public Attributes

- FILE * oFile
- std::string sName
- int * iDimensions
- int iStep

7.26.3 Detailed Description

Implements the abstract class needed for the plugins (.so) in charge of saving the results in an output file.

7.26.4 Constructor & Destructor Documentation

Reconfiguration::IFormat::IFormat ( )

Constructor of the IFormat class.

Returns

*this

Creates an object IFormat.

Reconfiguration::IFormat::~IFormat ( ) `[virtual]`

virtual destructor of the class IFormat.

Returns

void

Destroys the IFormat object.

7.26.5 Member Function Documentation

virtual void Reconfiguration::IFormat::save ([std::stringstream *]) [pure virtual]

Saves the user information into a file.

Parameters

in	*values*	User defined information.

Returns

void

Saves the user defined information into a file. This user defined function must meet the format specified by the user.

Implemented by user in the plugin file.

void Reconfiguration::IFormat::set ([std::string]sName, int *iDimension,* int * *iDimensions*)

Sets the properties of the object.

Parameters

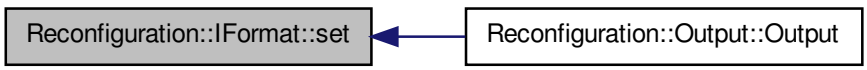
in	*sName*	Filename (without path).
in	*iDimension*	Dimensions number.
in	*iDimensions*	Length per each dimension.

Returns

void

Sets the values of the object properties.

Here is the caller graph for this function:



7.26.6 Member Data Documentation

int * Reconfiguration::IFormat::iDimensions

Length of the problem per dimension.

int Reconfiguration::IFormat::iStep

Step number since the beginning of the "simulation".

FILE * Reconfiguration::IFormat::oFile

File handler.

std::string Reconfiguration::IFormat::sName

File name.

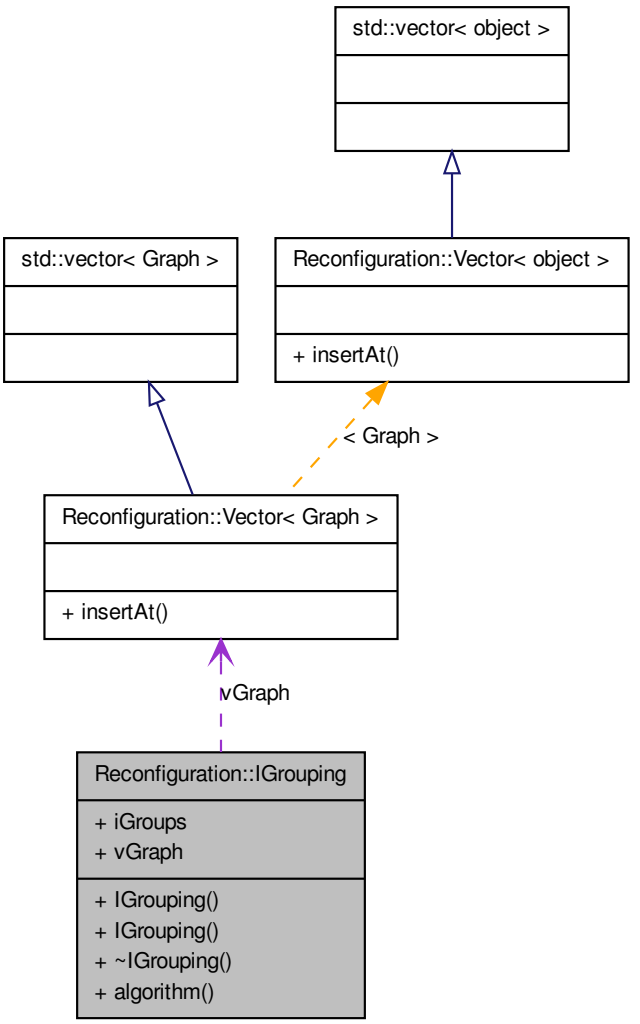
The documentation for this class was generated from the following files:

- IFormat.h
- IFormat.cpp

7.27 Reconfiguration::IGrouping Class Reference

`#include <IGrouping.h>`

Collaboration diagram for Reconfiguration::IGrouping:



7.27.1 Public Member Functions

- IGrouping ()

  Constructor of the IGrouping class.

- IGrouping (int)

Constructor of the IGrouping class.

- virtual ∼IGrouping ()

  virtual destructor of the class IGrouping.

- virtual void algorithm (Graph ∗)=0

  Groups the graph vertices depending on a user function.

7.27.2 Public Attributes

- int iGroups
- Vector< Graph > vGraph

7.27.3 Detailed Description

Implements the abstract class needed for the plugins (.so) in charge of grouping the cluster nodes.

7.27.4 Constructor & Destructor Documentation

Reconfiguration::IGrouping::IGrouping ( )

Constructor of the IGrouping class.

Returns

∗this

Creates an object IGrouping.

Reconfiguration::IGrouping::IGrouping ([int]iGroups)

Constructor of the IGrouping class.

Parameters

in	*iGroups*	Number of groups.

Returns

*this

Creates an object IGrouping setting the iGroups property.

Reconfiguration::IGrouping::~IGrouping ( ) [virtual]

virtual destructor of the class IGrouping.

Returns

void

Destroys the IGrouping object.

7.27.5 Member Function Documentation

virtual void Reconfiguration::IGrouping::algorithm ([Graph *]) [pure virtual]

Groups the graph vertices depending on a user function.

Parameters

in	*graph*	System graph.

Returns

void

Groups the system graph vertices as the user specifies.

Implemented by user in the plugin file.

7.27.6 Member Data Documentation

int Reconfiguration::IGrouping::iGroups

Number of groups.

Vector< Graph > Reconfiguration::IGrouping::vGraph

Vector of Graph components.

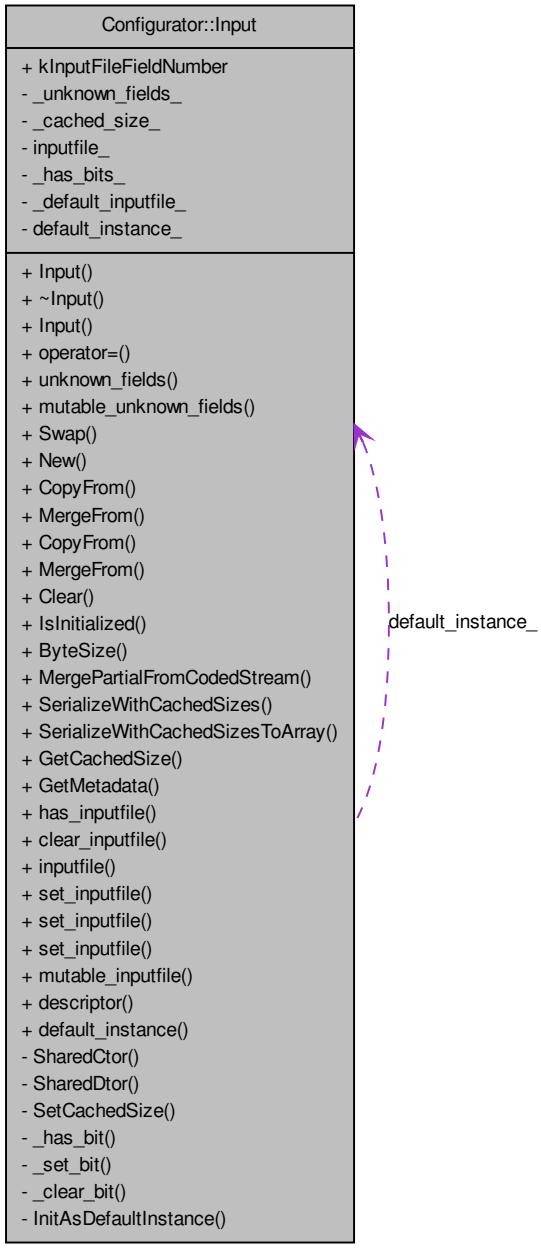
The documentation for this class was generated from the following files:

- IGrouping.h
- IGrouping.cpp

7.28 Configurator::Input Class Reference

`#include <confproblem.pb.h>`

Collaboration diagram for Configurator::Input:



7.28.1 Public Member Functions

- Input ()
- virtual ∼Input ()
- Input (const Input &from)
- Input & operator= (const Input &from)
- const ::google::protobuf::UnknownFieldSet & unknown_fields () const
- inline::google::protobuf::UnknownFieldSet ∗ mutable_unknown_fields ()
- void Swap (Input ∗other)
- Input ∗ New () const
- void CopyFrom (const ::google::protobuf::Message &from)
- void MergeFrom (const ::google::protobuf::Message &from)
- void CopyFrom (const Input &from)
- void MergeFrom (const Input &from)
- void Clear ()
- bool IsInitialized () const
- int ByteSize () const
- bool MergePartialFromCodedStream (::google::protobuf::io::CodedInputStream ∗input)
- void SerializeWithCachedSizes (::google::protobuf::io::CodedOutputStream ∗output) const
- ::google::protobuf::uint8 ∗ SerializeWithCachedSizesToArray (::google::protobuf::uint8 ∗output) const
- int GetCachedSize () const
- ::google::protobuf::Metadata GetMetadata () const
- bool has_inputfile () const
- void clear_inputfile ()
- const ::std::string & inputfile () const
- void set_inputfile (const ::std::string &value)
- void set_inputfile (const char ∗value)
- void set_inputfile (const char ∗value, size_t size)
- inline::std::string ∗ mutable_inputfile ()

7.28.2 Static Public Member Functions

- static const ::google::protobuf::Descriptor * descriptor ()
- static const Input & default_instance ()

7.28.3 Static Public Attributes

- static const int kInputFileFieldNumber = 1

7.28.4 Friends

- void protobuf_AddDesc_confproblem_2eproto ()
- void protobuf_AssignDesc_confproblem_2eproto ()
- void protobuf_ShutdownFile_confproblem_2eproto ()

7.28.5 Constructor & Destructor Documentation

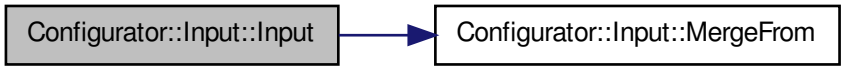
Configurator::Input::Input ( )

Here is the caller graph for this function:

Configurator::Input::~Input ( ) [virtual]

Configurator::Input::Input ([const Input &]from)

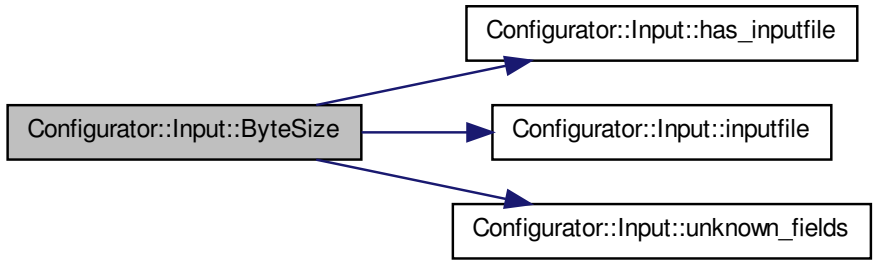
Here is the call graph for this function:



7.28.6 Member Function Documentation

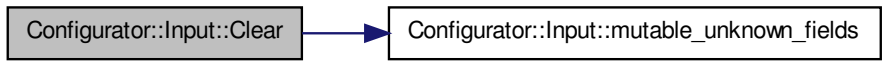
int Configurator::Input::ByteSize ( ) const

Here is the call graph for this function:

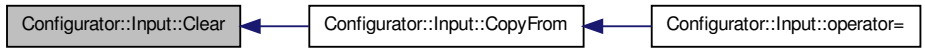


void Configurator::Input::Clear ( )

Here is the call graph for this function:



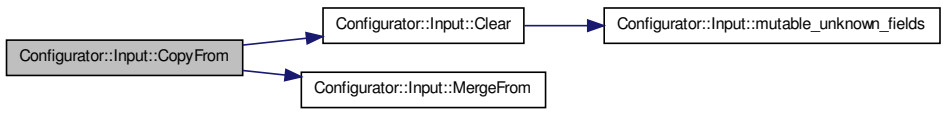
Here is the caller graph for this function:



void Configurator::Input::clear_inputfile ( ) `[inline]`

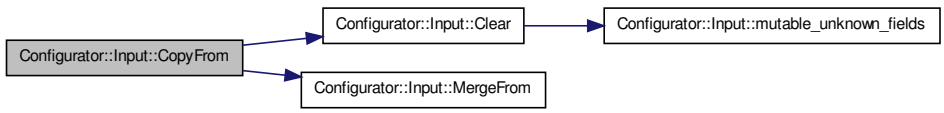
void Configurator::Input::CopyFrom ([const Input &]from)

Here is the call graph for this function:



void Configurator::Input::CopyFrom ([const ::google::protobuf::Message &]from)

Here is the call graph for this function:



Here is the caller graph for this function:

const Input & Configurator::Input::default_instance ( ) [static]

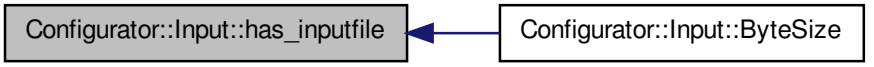
const ::google::protobuf::Descriptor * Configurator::Input::descriptor ( ) [static]

int Configurator::Input::GetCachedSize ( ) const [inline]

google::protobuf::Metadata Configurator::Input::GetMetadata ( ) const

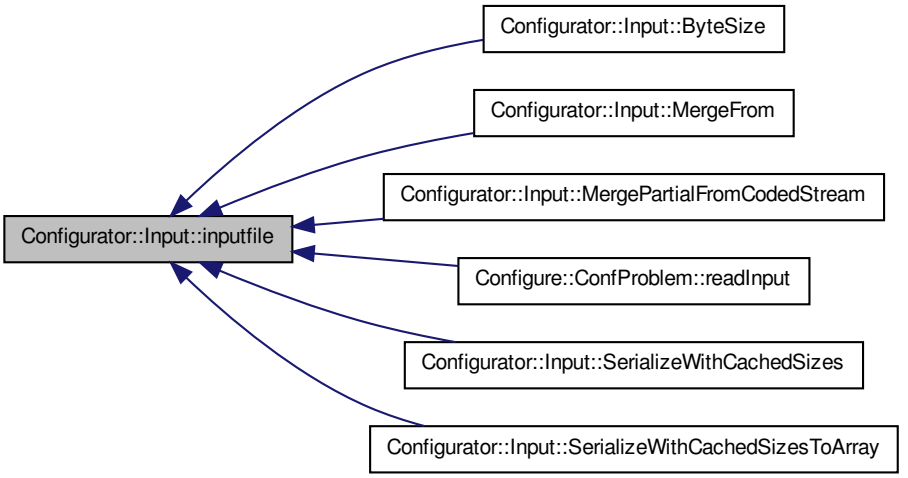
bool Configurator::Input::has_inputfile ( ) const [inline]

Here is the caller graph for this function:



const ::std::string & Configurator::Input::inputfile ( ) const [inline]

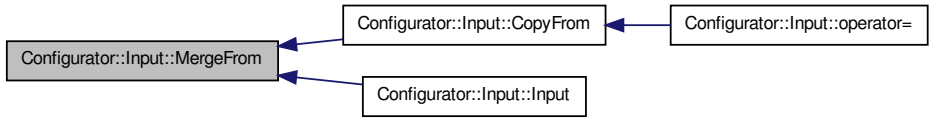
Here is the caller graph for this function:



bool Configurator::Input::IsInitialized ( ) const

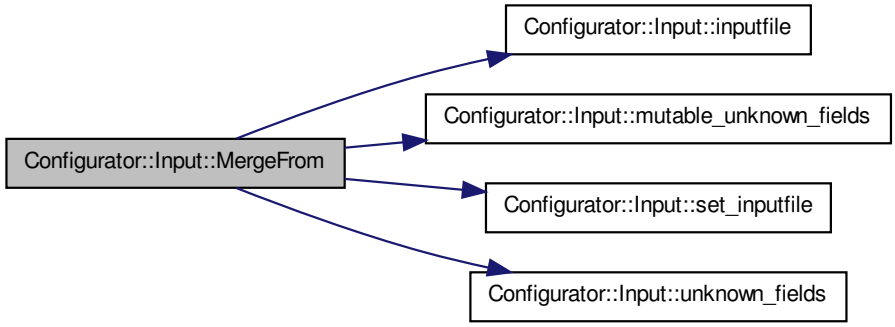
void Configurator::Input::MergeFrom ([const ::google::protobuf::Message &]from)

Here is the caller graph for this function:



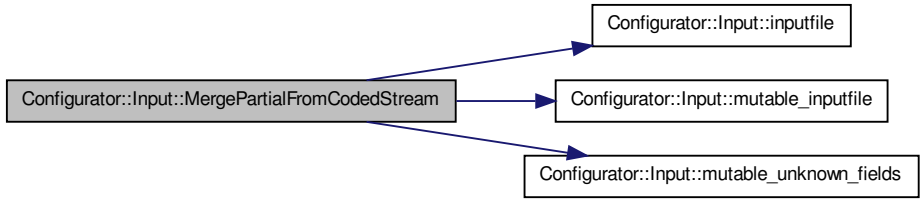
void Configurator::Input::MergeFrom ([const Input &]from)

Here is the call graph for this function:



bool Configurator::Input::MergePartialFromCodedStream ([::google::protobuf::io::CodedInputStream ∗]input)

Here is the call graph for this function:

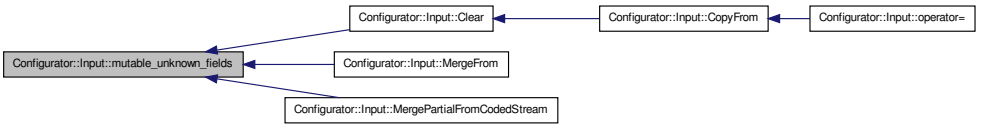


std::string ∗ Configurator::Input::mutable_inputfile ( ) [inline]

Here is the caller graph for this function:

inline ::google::protobuf::UnknownFieldSet∗ Configurator::Input::mutable_-unknown_fields ( ) [inline]

Here is the caller graph for this function:

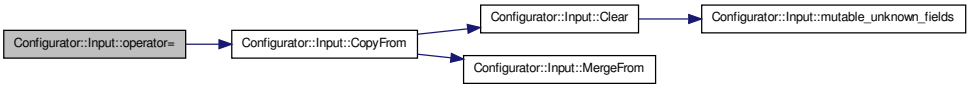


Input * Configurator::Input::New ( ) const

Here is the call graph for this function:

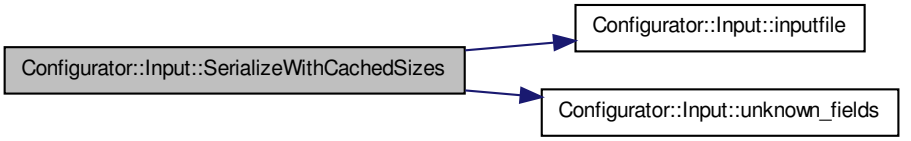
Input& Configurator::Input::operator= ([const Input &]from) `[inline]`

Here is the call graph for this function:



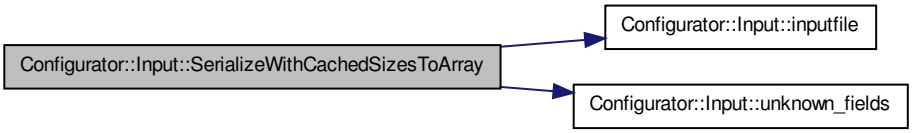
void Configurator::Input::SerializeWithCachedSizes ([::google::protobuf::io::CodedOutputStream *]output) const

Here is the call graph for this function:



google::protobuf::uint8 * Configurator::Input::SerializeWithCachedSizesToArray ([::google::protobuf::uint8 *]output) const

Here is the call graph for this function:



void Configurator::Input::set_inputfile ([const char *]value) `[inline]`

void Configurator::Input::set_inputfile ([const char *]value, size_t *size*) `[inline]`

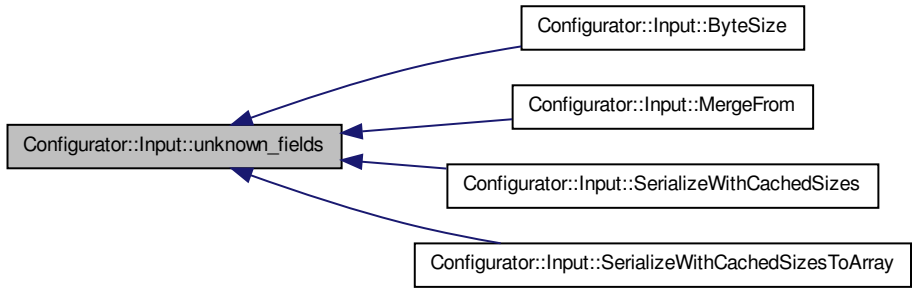
void Configurator::Input::set_inputfile ([const ::std::string &]value) `[inline]`

Here is the caller graph for this function:

void Configurator::Input::Swap ([Input *]other)

const ::google::protobuf::UnknownFieldSet& Configurator::Input::unknown_fields ( ) const [inline]

Here is the caller graph for this function:



7.28.7 Friends And Related Function Documentation

void protobuf_AddDesc_confproblem_2eproto ( ) [friend]

void protobuf_AssignDesc_confproblem_2eproto ( ) [friend]

void protobuf_ShutdownFile_confproblem_2eproto ( ) [friend]

7.28.8 Member Data Documentation

const int Configurator::Input::kInputFileFieldNumber = 1 [static]

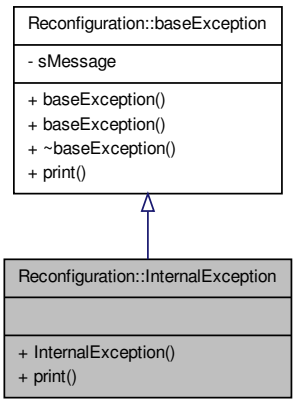
The documentation for this class was generated from the following files:

- confproblem.pb.h
- confproblem.pb.cc

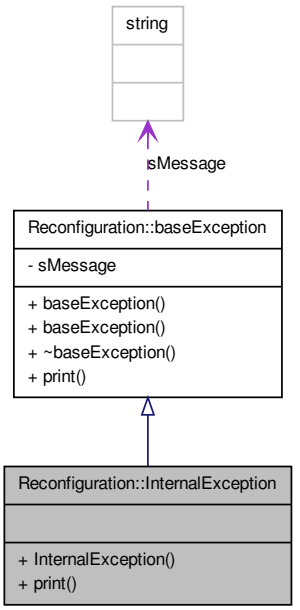
7.29 Reconfiguration::InternalException Class Reference

`#include <Exception.h>`

Inheritance diagram for Reconfiguration::InternalException:



Collaboration diagram for Reconfiguration::InternalException:



7.29.1 Public Member Functions

- InternalException (std::string)

  Constructor of the InternalException class.

- void print ()

  Prints the InternalException exception.

7.29.2 Detailed Description

Exception thrown if an internal error occurs.

7.29.3 Constructor & Destructor Documentation

Reconfiguration::InternalException::InternalException ([std::string]sText)

Constructor of the InternalException class.

Parameters

in	*sText*	Internal exception text.

Returns

∗this

Creates an object InternalException calling the baseException constructor and sets the message of the exception.

7.29.4 Member Function Documentation

void Reconfiguration::InternalException::print ( )

Prints the InternalException exception.

Returns

void

Prints the exception by printing its message.

Reimplemented from Reconfiguration::baseException.

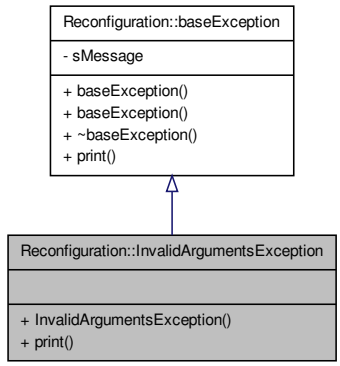
The documentation for this class was generated from the following files:

- Exception.h
- Exception.cpp

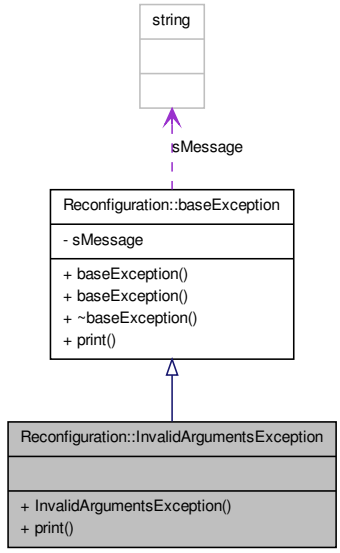
7.30 Reconfiguration::InvalidArgumentsException Class Reference

`#include <Exception.h>`

Inheritance diagram for Reconfiguration::InvalidArgumentsException:



Collaboration diagram for Reconfiguration::InvalidArgumentsException:



7.30.1 Public Member Functions

- InvalidArgumentsException (std::string)

  Constructor of the InvalidArgumentsException class.

- void print ()

  Prints the InvalidArgumentsException exception.

7.30.2 Detailed Description

Exception thrown if a function receives an invalid argument.

7.30.3 Constructor & Destructor Documentation

Reconfiguration::InvalidArgumentsException::InvalidArgumentsException ([std::string]sText)

Constructor of the InvalidArgumentsException class.

Parameters

in	*sText*	Invalid argument text.

Returns

∗this

Creates an object InvalidArgumentsException calling the baseException constructor and sets the message of the exception.

7.30.4 Member Function Documentation

void Reconfiguration::InvalidArgumentsException::print ( )

Prints the InvalidArgumentsException exception.

Returns

void

Prints the exception by printing its message.

Reimplemented from Reconfiguration::baseException.

The documentation for this class was generated from the following files:

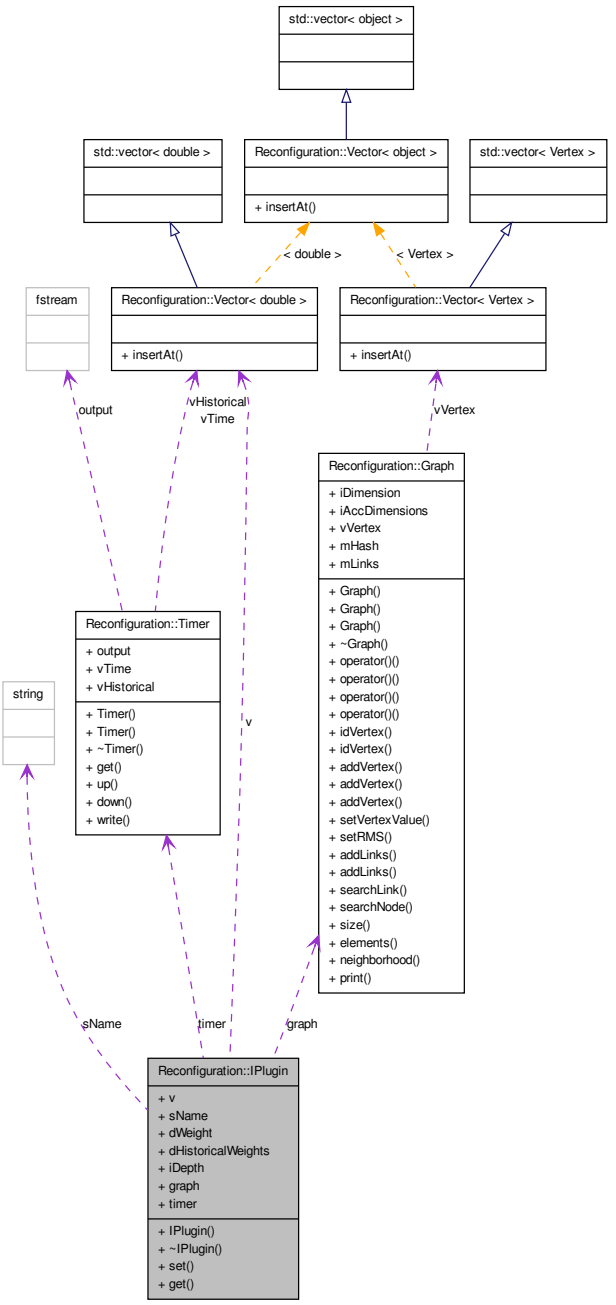
- Exception.h

- Exception.cpp

7.31 Reconfiguration::IPlugin Class Reference

`#include <IPlugin.h>`

Collaboration diagram for Reconfiguration::IPlugin:



7.31.1 Public Member Functions

- IPlugin ()

  Constructor of the IPlugin class.

- virtual ∼IPlugin ()

  virtual destructor of the class IPlugin.

- void set (std::string, double, int, double ∗, Graph ∗, Timer ∗)

  Sets the properties of the object.

- virtual void get ()=0

  Gets the results obtained by the plugin.

7.31.2 Public Attributes

- Vector< double > v
- std::string sName
- double dWeight
- double ∗ dHistoricalWeights
- int iDepth
- Graph ∗ graph
- Timer ∗ timer

7.31.3 Detailed Description

Implements the abstract class needed for the plugins (.so) in charge of calculating the load-balancing.

7.31.4 Constructor & Destructor Documentation

Reconfiguration::IPlugin::IPlugin ( )

Constructor of the IPlugin class.

Returns

∗this

Creates an object IPlugin.

Reconfiguration::IPlugin::∼IPlugin ( ) [virtual]

virtual destructor of the class IPlugin.

Returns

void

Destroys the IPlugin object.

7.31.5 Member Function Documentation

virtual void Reconfiguration::IPlugin::get ( ) [pure virtual]

Gets the results obtained by the plugin.

Returns

void

Reads the values for the loadbalancing obtained by the plugins. Implemented by user in the plugin file.

void Reconfiguration::IPlugin::set ([std::string]sName, double *dWeight,* int *iDepth,* double ∗ *dHistoricalWeights,* Graph ∗ *graph,* Timer ∗ *timer*)

Sets the properties of the object.

Parameters

in	*sName*	Loadbalancing plugin name.
in	*dWeight*	Plugin weight.
in	*iDepth*	Length of the valid historical values.
in	*dHistoricalWeights*	Weight of each historical value, if the plugin is based on historical results.
in	*graph*	System graph.
in	*timer*	Timer to time the execution time of each slave.

Returns

void

Sets the values of the object properties.

Here is the caller graph for this function:

7.31.6 Member Data Documentation

double * Reconfiguration::IPlugin::dHistoricalWeights

Weight of each historical value, if the plugin is based on historical results.

double Reconfiguration::IPlugin::dWeight

Plugin weight.

Graph * Reconfiguration::IPlugin::graph

System graph.

int Reconfiguration::IPlugin::iDepth

Length of the valid historical values.

std::string Reconfiguration::IPlugin::sName

Loadbalancing plugin name.

Timer * Reconfiguration::IPlugin::timer

Timer to time the execution time of each slave.

Vector< double > Reconfiguration::IPlugin::v

Vector where the plugins returns its calculations.

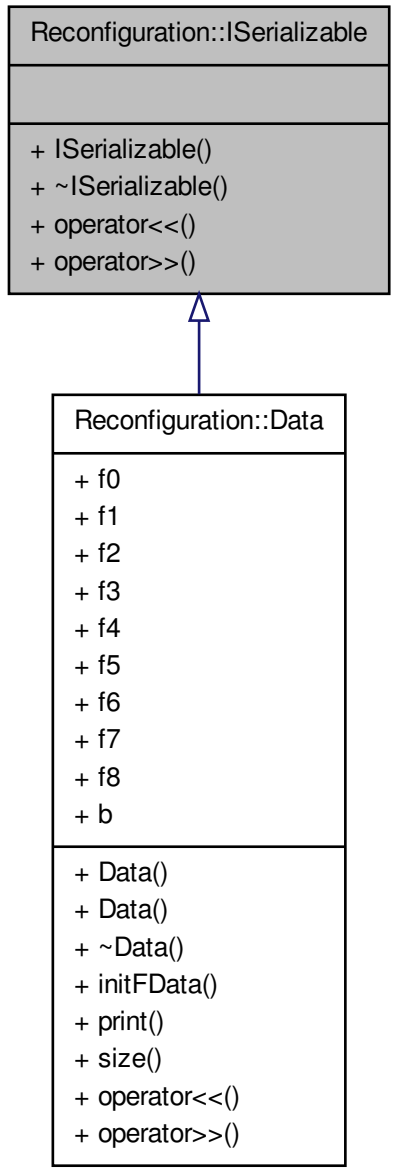
The documentation for this class was generated from the following files:

- IPlugin.h
- IPlugin.cpp

7.32 Reconfiguration::ISerializable Class Reference

`#include <ISerializable.h>`

Inheritance diagram for Reconfiguration::ISerializable:



7.32.1 Public Member Functions

- ISerializable ()

  Constructor of the ISerializable class.

- virtual ∼ISerializable ()

  virtual destructor of the class ISerializable.

- virtual std::stringstream & operator<< (std::stringstream &)=0

  Deserialization of an object.

- virtual std::stringstream & operator>> (std::stringstream &)=0

  Serialization of an object.

7.32.2 Detailed Description

Implements the abstract class needed for serialize and deserialize the user Data object.

7.32.3 Constructor & Destructor Documentation

Reconfiguration::ISerializable::ISerializable ( )

Constructor of the ISerializable class.

Returns

∗this

Creates an object ISerializable.

Reconfiguration::ISerializable::∼ISerializable ( ) `[virtual]`

virtual destructor of the class ISerializable.

Returns

void

Destroys the ISerializable object.

7.32.4 Member Function Documentation

virtual std::stringstream& Reconfiguration::ISerializable::operator<< ([std::stringstream &]) [pure virtual]

Deserialization of an object.

Returns

std::stringstream

Deserialization of an object. Returns the modified std::stringstream. Implemented by user in the Data file.

Implemented in Reconfiguration::Data.

virtual std::stringstream& Reconfiguration::ISerializable::operator>> ([std::stringstream &]) [pure virtual]

Serialization of an object.

Returns

std::stringstream

Serialization of an object. Returns the modified std::stringstream. Implemented by user in the Data file.

Implemented in Reconfiguration::Data.

The documentation for this class was generated from the following files:

- ISerializable.h
- ISerializable.cpp

7.33 Reconfiguration::ISorting Class Reference

`#include <ISorting.h>`

7.33.1 Public Member Functions

- ISorting ()

  Constructor of the ISorting class.

- virtual ∼ISorting ()

  virtual destructor of the class ISorting.

- virtual void sort (Graph ∗)=0

  Calculates the arragement of the vertices of the graph.

- void print ()

  prints the vertex arragement.

7.33.2 Public Attributes

- int iSize
- int ∗ iSort

7.33.3 Detailed Description

Implements the abstract class needed for the plugins (.so) in charge of calculating the graph decomposition.

7.33.4 Constructor & Destructor Documentation

Reconfiguration::ISorting::ISorting ( )

Constructor of the ISorting class.

Returns

*this

Creates an object ISorting.

Reconfiguration::ISorting::~ISorting ( ) [virtual]

virtual destructor of the class ISorting.

Returns

void

Destroys the ISorting object.

7.33.5 Member Function Documentation

void Reconfiguration::ISorting::print ( )

prints the vertex arragement.

Returns

void

Prints the ISorting object printing the arramgent vector.

Here is the caller graph for this function:

virtual void Reconfiguration::ISorting::sort ([Graph *]) [pure virtual]

Calculates the arragement of the vertices of the graph.

Parameters

in	*graph*	Application graph.

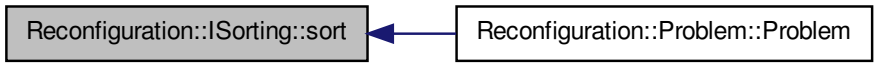
Returns

void

Calculates the arragement of the vertices of the graph.

Implemented by user in the plugin file.

Here is the caller graph for this function:



7.33.6 Member Data Documentation

int Reconfiguration::ISorting::iSize

Number of domains that must be created.

int Reconfiguration::ISorting::iSort

Arragement of vertices identifiers.

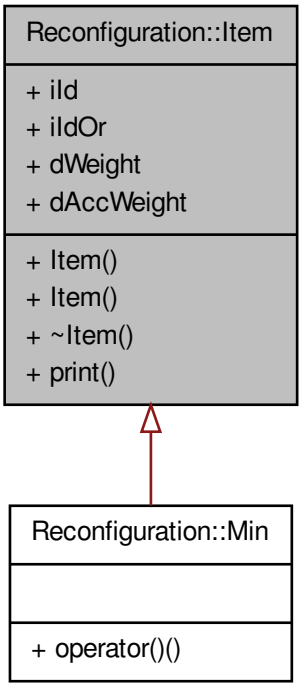
The documentation for this class was generated from the following files:

- ISorting.h
- ISorting.cpp

7.34 Reconfiguration::Item Class Reference

`#include <IGrouping.h>`

Inheritance diagram for Reconfiguration::Item:



7.34.1 Public Member Functions

- Item ()

  Constructor of the Item class.

- Item (int, int, double, double)

  Constructor of the Item class.

- virtual ~Item ()

  virtual destructor of the class Item.

- void print ()

  prints the Item object.

7.34.2 Public Attributes

- int iId
- int iIdOr
- double dWeight
- double dAccWeight

7.34.3 Detailed Description

Structure for allocate the identifiers and the weight of vertices while visiting them to implement the grouping algorithm.

7.34.4 Constructor & Destructor Documentation

Reconfiguration::Item::Item ( )

Constructor of the Item class.

Returns

∗this

Creates an object Item.

Reconfiguration::Item::Item ([int]id, int *idOr,* double *dWeight,* double *dAccWeight*)

Constructor of the Item class.

Parameters

in	*id*	Vertex identifier.
in	*idOr*	Source Vertex identifier.
in	*dWeight*	Vertex weight.
in	*dAccWeight*	Accumulated weight from the beginning of the path.

Returns

∗this

Creates an object Item setting ist properties.

Reconfiguration::Item::~Item ( ) [virtual]

virtual destructor of the class Item.

Returns

void

Destroys the Item object.

7.34.5 Member Function Documentation

void Reconfiguration::Item::print ( )

prints the Item object.

Returns

void

Prints the Item object.

7.34.6 Member Data Documentation

int Reconfiguration::Item::dAccWeight

Accumulated weight from the beginning of the path.

int Reconfiguration::Item::dWeight

Vertex weight.

int Reconfiguration::Item::iId

Vertex identifier.

int Reconfiguration::Item::iIdOr

Source Vertex identifier.

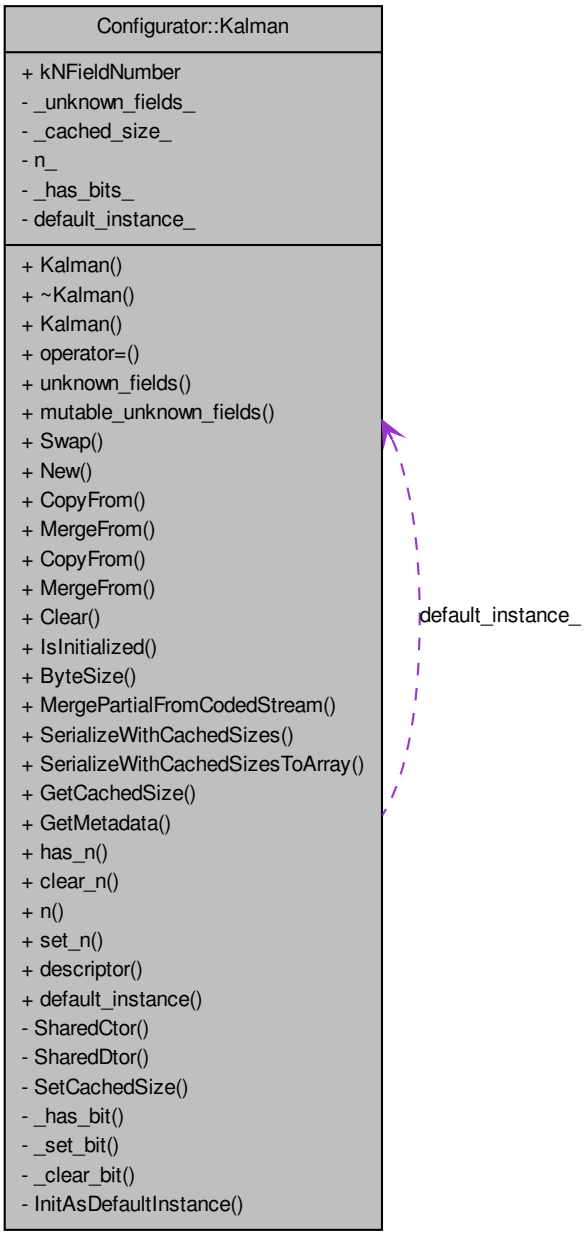
The documentation for this class was generated from the following files:

- IGrouping.h
- IGrouping.cpp

7.35 Configurator::Kalman Class Reference

`#include <confpartitioner.pb.h>`

Collaboration diagram for Configurator::Kalman:



7.35.1 Public Member Functions

- Kalman ()
- virtual ∼Kalman ()
- Kalman (const Kalman &from)
- Kalman & operator= (const Kalman &from)
- const ::google::protobuf::UnknownFieldSet & unknown_fields () const
- inline::google::protobuf::UnknownFieldSet ∗ mutable_unknown_fields ()
- void Swap (Kalman ∗other)
- Kalman ∗ New () const
- void CopyFrom (const ::google::protobuf::Message &from)
- void MergeFrom (const ::google::protobuf::Message &from)
- void CopyFrom (const Kalman &from)
- void MergeFrom (const Kalman &from)
- void Clear ()
- bool IsInitialized () const
- int ByteSize () const
- bool MergePartialFromCodedStream (::google::protobuf::io::CodedInputStream ∗input)
- void SerializeWithCachedSizes (::google::protobuf::io::CodedOutputStream ∗output) const
- ::google::protobuf::uint8 ∗ SerializeWithCachedSizesToArray (::google::protobuf::uint8 ∗output) const
- int GetCachedSize () const
- ::google::protobuf::Metadata GetMetadata () const
- bool has_n () const
- void clear_n ()
- inline::google::protobuf::int32 n () const
- void set_n (::google::protobuf::int32 value)

7.35.2 Static Public Member Functions

- static const ::google::protobuf::Descriptor ∗ descriptor ()
- static const Kalman & default_instance ()

7.35.3 Static Public Attributes

- static const int kNFieldNumber = 1

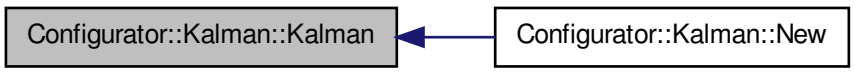
7.35.4 Friends

- void protobuf_AddDesc_confpartitioner_2eproto ()
- void protobuf_AssignDesc_confpartitioner_2eproto ()
- void protobuf_ShutdownFile_confpartitioner_2eproto ()

7.35.5 Constructor & Destructor Documentation

Configurator::Kalman::Kalman ( )

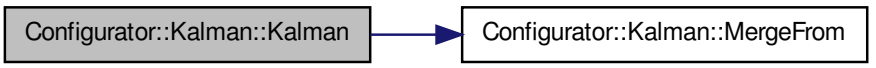
Here is the caller graph for this function:



Configurator::Kalman::~Kalman ( ) [virtual]

Configurator::Kalman::Kalman ([const Kalman &]from)

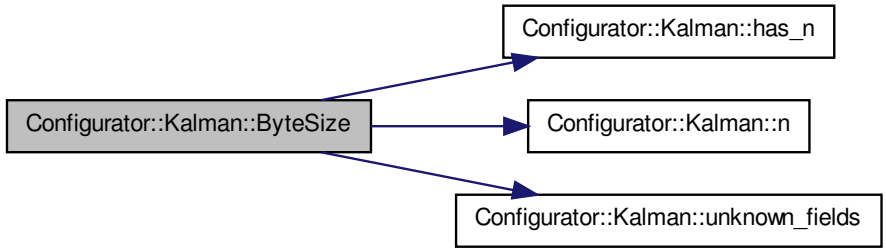
Here is the call graph for this function:



7.35.6 Member Function Documentation

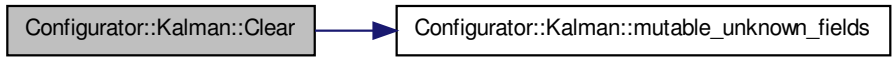
int Configurator::Kalman::ByteSize ( ) const

Here is the call graph for this function:

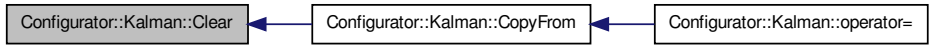


void Configurator::Kalman::Clear ( )

Here is the call graph for this function:



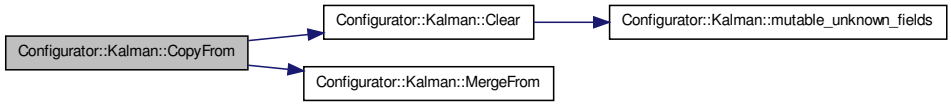
Here is the caller graph for this function:



void Configurator::Kalman::clear_n ( ) [inline]

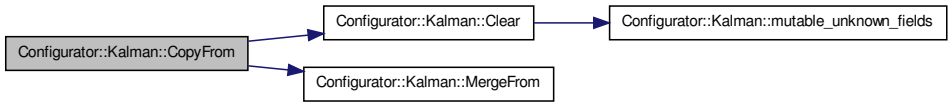
void Configurator::Kalman::CopyFrom ([const Kalman &]from)

Here is the call graph for this function:



void Configurator::Kalman::CopyFrom ([const ::google::protobuf::Message &]from)

Here is the call graph for this function:



Here is the caller graph for this function:

const Kalman & Configurator::Kalman::default_instance ( ) [static]

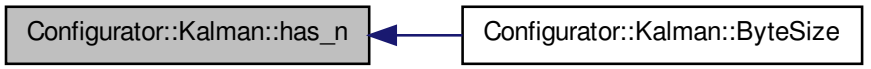
const ::google::protobuf::Descriptor * Configurator::Kalman::descriptor ( ) [static]

int Configurator::Kalman::GetCachedSize ( ) const [inline]

google::protobuf::Metadata Configurator::Kalman::GetMetadata ( ) const

bool Configurator::Kalman::has_n ( ) const [inline]

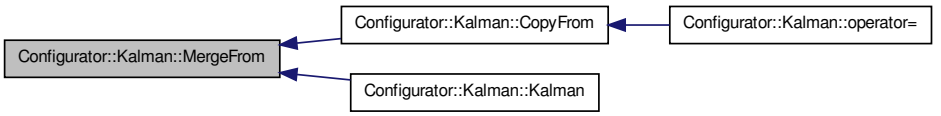
Here is the caller graph for this function:



bool Configurator::Kalman::IsInitialized ( ) const

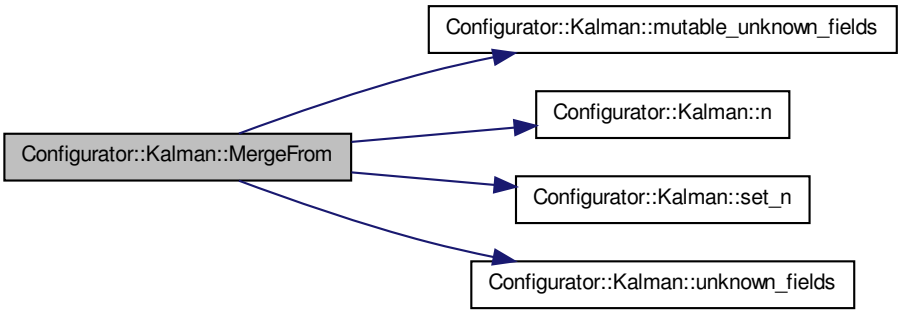
void Configurator::Kalman::MergeFrom ([const ::google::protobuf::Message &]from)

Here is the caller graph for this function:



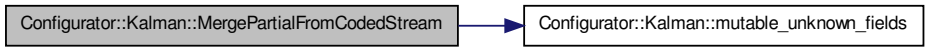
void Configurator::Kalman::MergeFrom ([const Kalman &]from)

Here is the call graph for this function:



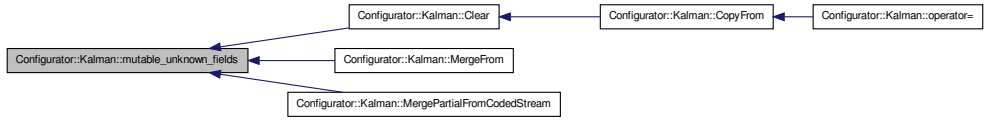
bool Configurator::Kalman::MergePartialFromCodedStream ([::google::protobuf::io::CodedInputStream *]input)

Here is the call graph for this function:



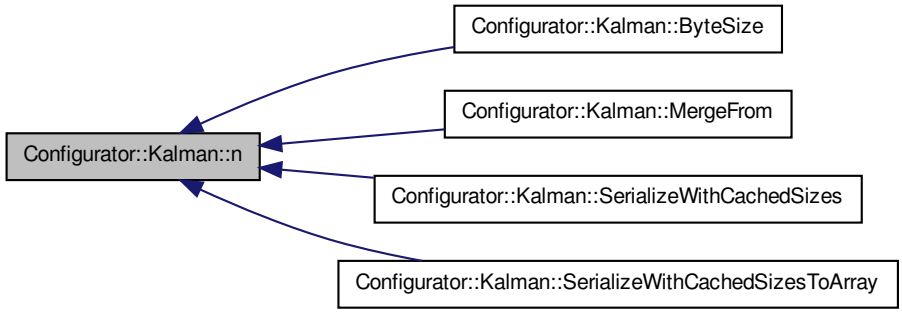
inline ::google::protobuf::UnknownFieldSet* Configurator::Kalman::mutable_-unknown_fields ( ) [inline]

Here is the caller graph for this function:



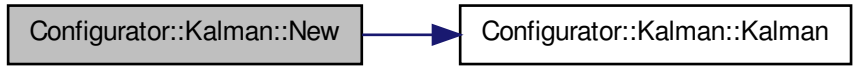
google::protobuf::int32 Configurator::Kalman::n ( ) const [inline]

Here is the caller graph for this function:



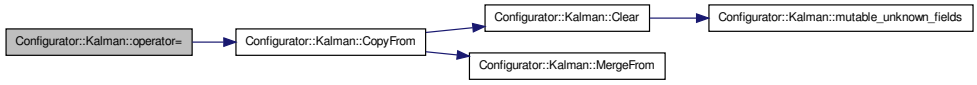
Kalman * Configurator::Kalman::New ( ) const

Here is the call graph for this function:



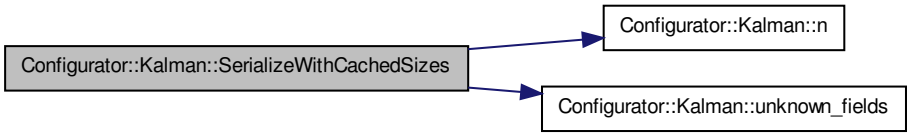
Kalman& Configurator::Kalman::operator= ([const Kalman &]from) [inline]

Here is the call graph for this function:



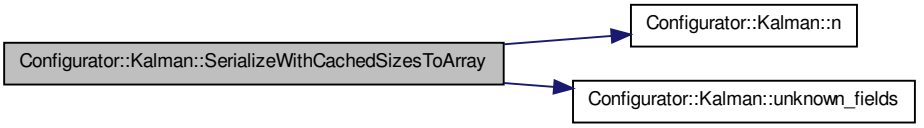
void Configurator::Kalman::SerializeWithCachedSizes ([::google::protobuf::io::CodedOutputStream *]output) const

Here is the call graph for this function:



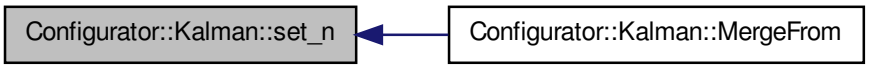
google::protobuf::uint8 * Configurator::Kalman::SerializeWithCachedSizesToArray ([::google::protobuf::uint8 *]output) const

Here is the call graph for this function:



void Configurator::Kalman::set_n ([::google::protobuf::int32]value) `[inline]`

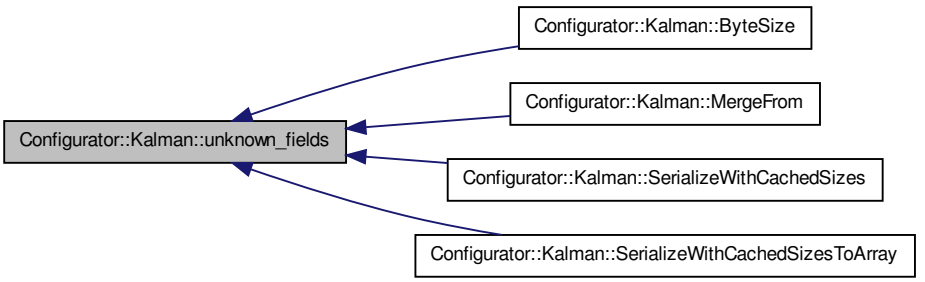
Here is the caller graph for this function:



void Configurator::Kalman::Swap ([Kalman *]other)

const ::google::protobuf::UnknownFieldSet& Configurator::Kalman::unknown_fields ( ) const `[inline]`

Here is the caller graph for this function:



7.35.7 Friends And Related Function Documentation

void protobuf_AddDesc_confpartitioner_2eproto ( ) [friend]

void protobuf_AssignDesc_confpartitioner_2eproto ( ) [friend]

void protobuf_ShutdownFile_confpartitioner_2eproto ( ) [friend]

7.35.8 Member Data Documentation

const int Configurator::Kalman::kNFieldNumber = 1 [static]

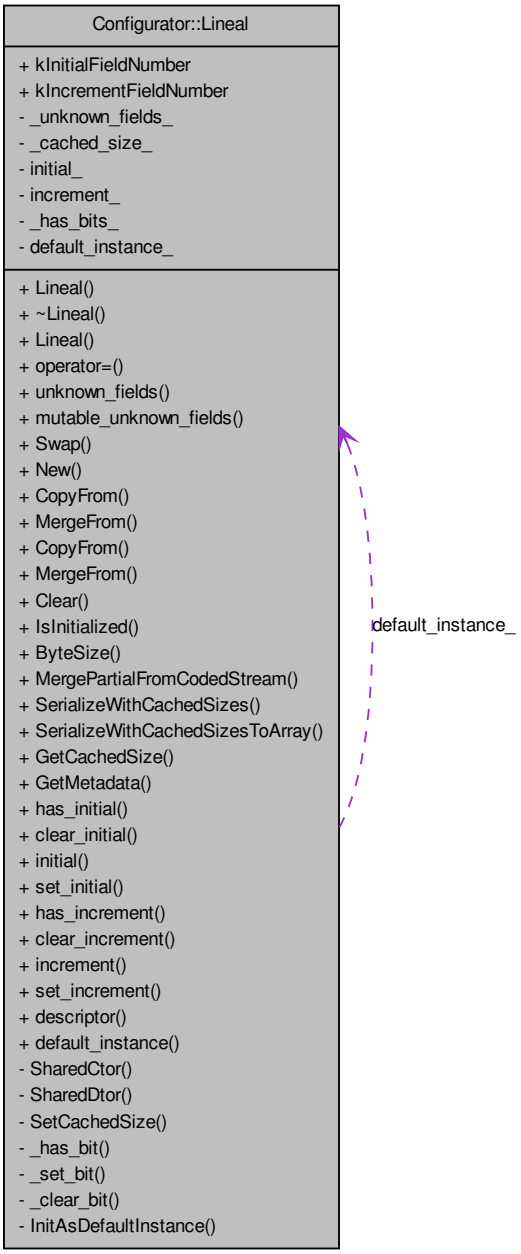
The documentation for this class was generated from the following files:

- confpartitioner.pb.h
- confpartitioner.pb.cc

7.36 Configurator::Lineal Class Reference

`#include <confpartitioner.pb.h>`

Collaboration diagram for Configurator::Lineal:



7.36.1 Public Member Functions

- Lineal ()
- virtual ∼Lineal ()
- Lineal (const Lineal &from)
- Lineal & operator= (const Lineal &from)
- const ::google::protobuf::UnknownFieldSet & unknown_fields () const
- inline::google::protobuf::UnknownFieldSet ∗ mutable_unknown_fields ()
- void Swap (Lineal ∗other)
- Lineal ∗ New () const
- void CopyFrom (const ::google::protobuf::Message &from)
- void MergeFrom (const ::google::protobuf::Message &from)
- void CopyFrom (const Lineal &from)
- void MergeFrom (const Lineal &from)
- void Clear ()
- bool IsInitialized () const
- int ByteSize () const
- bool MergePartialFromCodedStream (::google::protobuf::io::CodedInputStream ∗input)
- void SerializeWithCachedSizes (::google::protobuf::io::CodedOutputStream ∗output) const
- ::google::protobuf::uint8 ∗ SerializeWithCachedSizesToArray (::google::protobuf::uint8 ∗output) const
- int GetCachedSize () const
- ::google::protobuf::Metadata GetMetadata () const
- bool has_initial () const
- void clear_initial ()
- inline::google::protobuf::int32 initial () const
- void set_initial (::google::protobuf::int32 value)
- bool has_increment () const
- void clear_increment ()
- inline::google::protobuf::int32 increment () const
- void set_increment (::google::protobuf::int32 value)

7.36.2 Static Public Member Functions

- static const ::google::protobuf::Descriptor * descriptor ()
- static const Lineal & default_instance ()

7.36.3 Static Public Attributes

- static const int kInitialFieldNumber = 1
- static const int kIncrementFieldNumber = 2

7.36.4 Friends

- void protobuf_AddDesc_confpartitioner_2eproto ()
- void protobuf_AssignDesc_confpartitioner_2eproto ()
- void protobuf_ShutdownFile_confpartitioner_2eproto ()

7.36.5 Constructor & Destructor Documentation

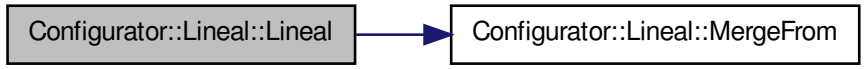
Configurator::Lineal::Lineal ( )

Here is the caller graph for this function:

Configurator::Lineal::~Lineal ( ) [virtual]

Configurator::Lineal::Lineal ([const Lineal &]from)

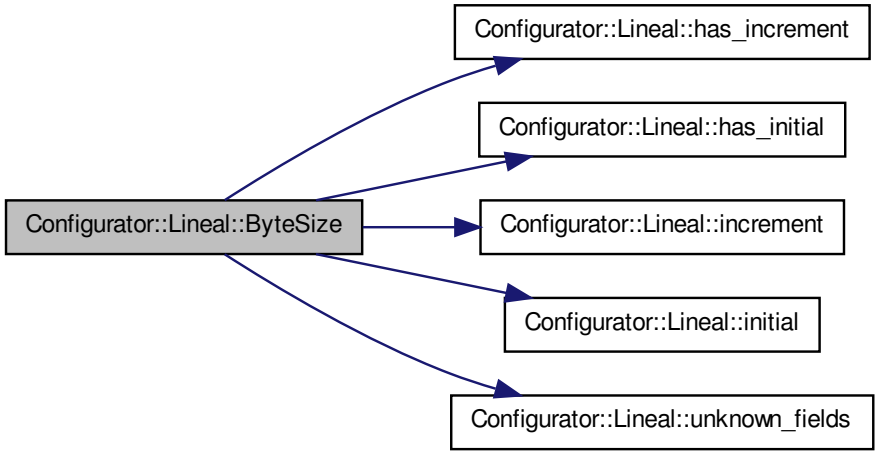
Here is the call graph for this function:



7.36.6 Member Function Documentation

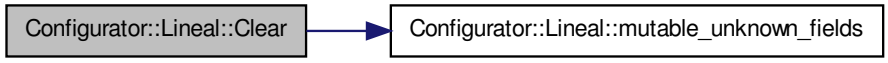
int Configurator::Lineal::ByteSize ( ) const

Here is the call graph for this function:

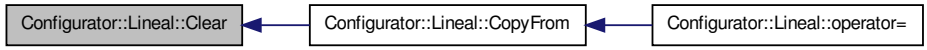


void Configurator::Lineal::Clear ( )

Here is the call graph for this function:



Here is the caller graph for this function:

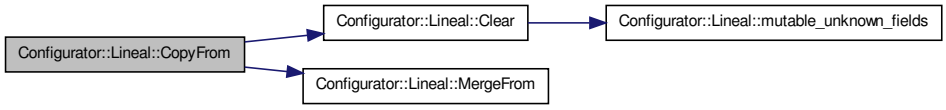


void Configurator::Lineal::clear_increment ( ) `[inline]`

void Configurator::Lineal::clear_initial ( ) `[inline]`

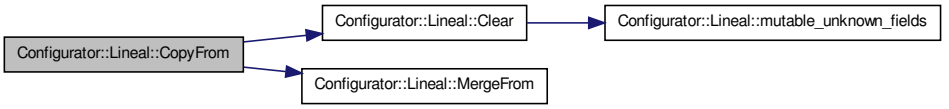
void Configurator::Lineal::CopyFrom ([const Lineal &]from)

Here is the call graph for this function:



void Configurator::Lineal::CopyFrom ([const ::google::protobuf::Message &]from)

Here is the call graph for this function:



Here is the caller graph for this function:

const Lineal & Configurator::Lineal::default_instance ( ) [static]

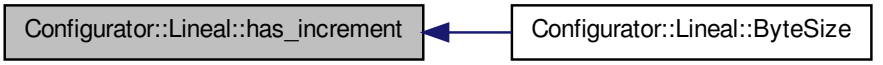
const ::google::protobuf::Descriptor * Configurator::Lineal::descriptor ( ) [static]

int Configurator::Lineal::GetCachedSize ( ) const [inline]

google::protobuf::Metadata Configurator::Lineal::GetMetadata ( ) const

bool Configurator::Lineal::has_increment ( ) const [inline]

Here is the caller graph for this function:

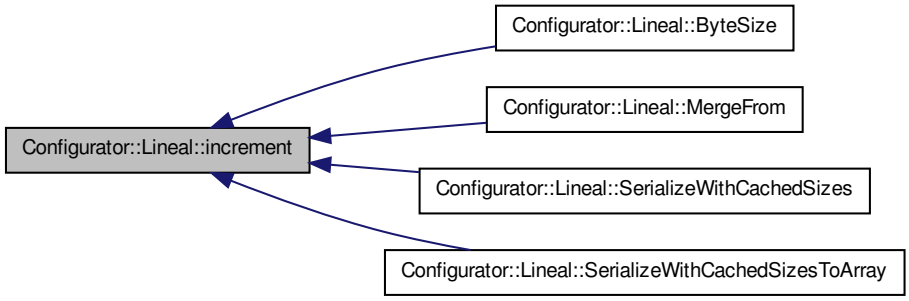


bool Configurator::Lineal::has_initial ( ) const [inline]

Here is the caller graph for this function:

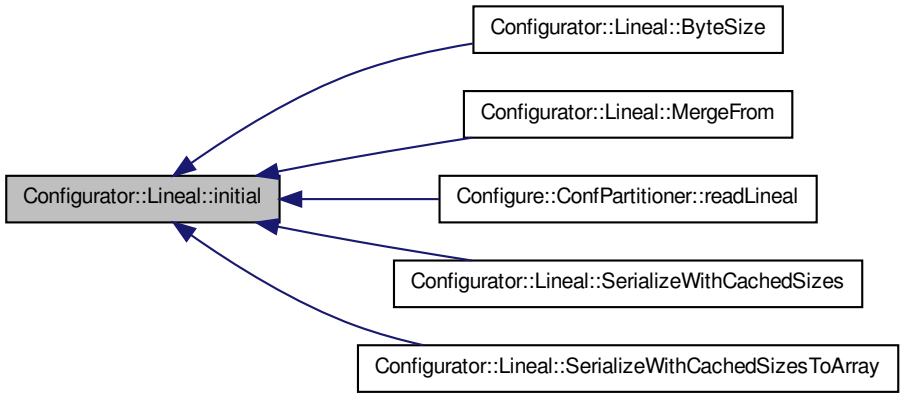
google::protobuf::int32 Configurator::Lineal::increment ( ) const [inline]

Here is the caller graph for this function:



google::protobuf::int32 Configurator::Lineal::initial ( ) const `[inline]`

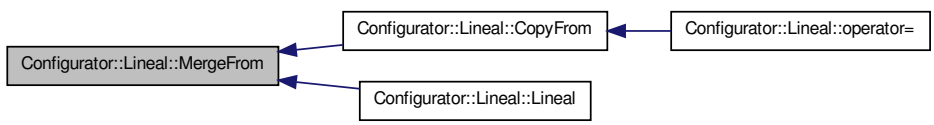
Here is the caller graph for this function:



bool Configurator::Lineal::IsInitialized ( ) const

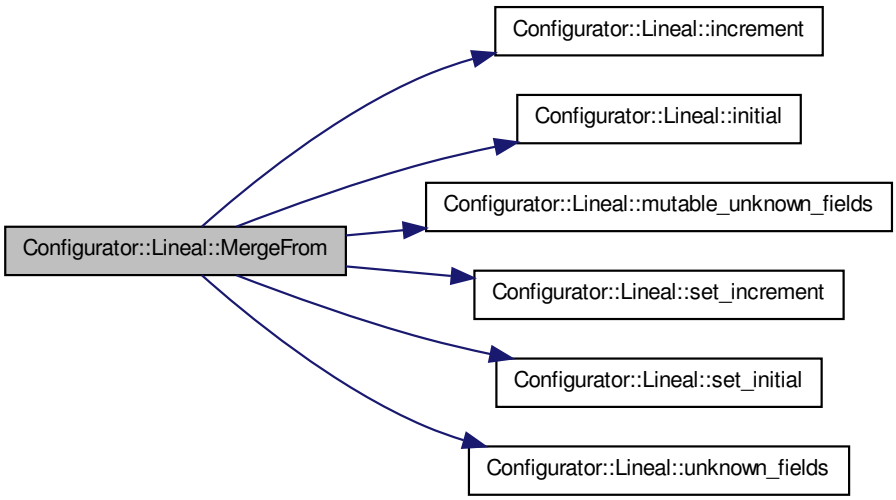
void Configurator::Lineal::MergeFrom ([const ::google::protobuf::Message &]from)

Here is the caller graph for this function:



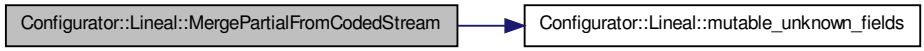
void Configurator::Lineal::MergeFrom ([const Lineal &]from)

Here is the call graph for this function:



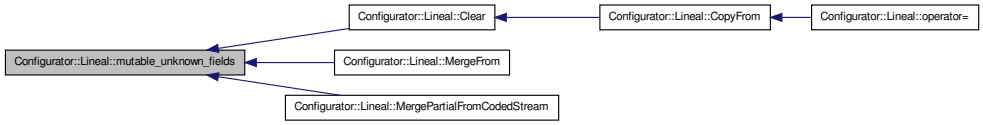
bool Configurator::Lineal::MergePartialFromCodedStream ([::google::protobuf::io::CodedInputStream ∗]input)

Here is the call graph for this function:



inline ::google::protobuf::UnknownFieldSet∗ Configurator::Lineal::mutable_-unknown_fields ( ) [inline]

Here is the caller graph for this function:

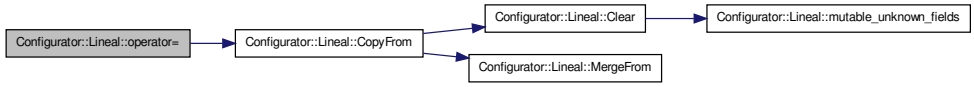


Lineal ∗ Configurator::Lineal::New ( ) const

Here is the call graph for this function:

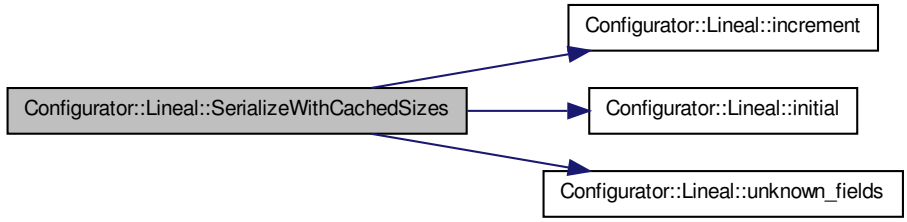
Lineal& Configurator::Lineal::operator= ([const Lineal &]from) [inline]

Here is the call graph for this function:



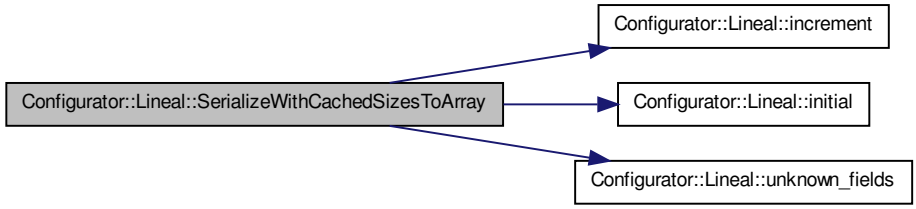
void Configurator::Lineal::SerializeWithCachedSizes ([::google::protobuf::io::CodedOutputStream *]output) const

Here is the call graph for this function:



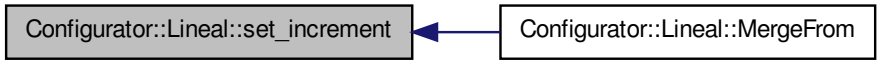
google::protobuf::uint8 * Configurator::Lineal::SerializeWithCachedSizesToArray ([::google::protobuf::uint8 *]output) const

Here is the call graph for this function:



void Configurator::Lineal::set_increment ([::google::protobuf::int32]value) `[inline]`

Here is the caller graph for this function:



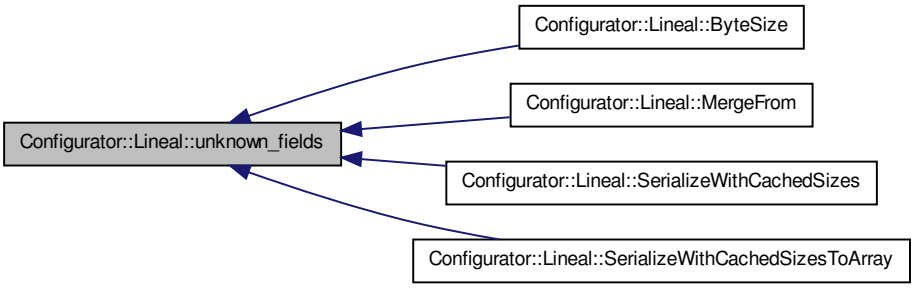
void Configurator::Lineal::set_initial ([::google::protobuf::int32]value) `[inline]`

Here is the caller graph for this function:

void Configurator::Lineal::Swap ([Lineal *]other)

const ::google::protobuf::UnknownFieldSet& Configurator::Lineal::unknown_fields ( ) const [inline]

Here is the caller graph for this function:



7.36.7 Friends And Related Function Documentation

void protobuf_AddDesc_confpartitioner_2eproto ( ) [friend]

void protobuf_AssignDesc_confpartitioner_2eproto ( ) [friend]

void protobuf_ShutdownFile_confpartitioner_2eproto ( ) [friend]

7.36.8 Member Data Documentation

const int Configurator::Lineal::kIncrementFieldNumber = 2 [static]

const int Configurator::Lineal::kInitialFieldNumber = 1 [static]

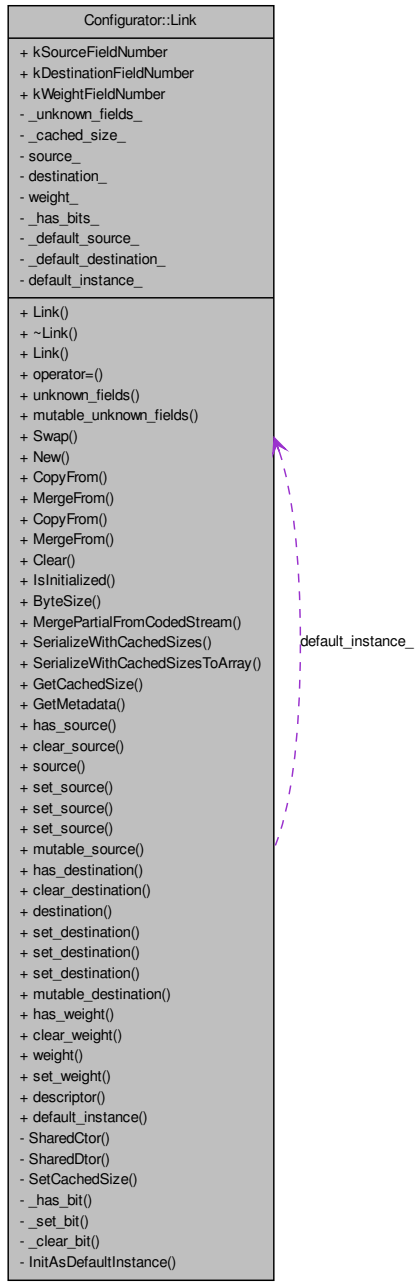
The documentation for this class was generated from the following files:

- confpartitioner.pb.h
- confpartitioner.pb.cc

7.37 Configurator::Link Class Reference

`#include <confcluster.pb.h>`

Collaboration diagram for Configurator::Link:



7.37.1 Public Member Functions

- Link ()
- virtual ∼Link ()
- Link (const Link &from)
- Link & operator= (const Link &from)
- const ::google::protobuf::UnknownFieldSet & unknown_fields () const
- inline::google::protobuf::UnknownFieldSet ∗ mutable_unknown_fields ()
- void Swap (Link ∗other)
- Link ∗ New () const
- void CopyFrom (const ::google::protobuf::Message &from)
- void MergeFrom (const ::google::protobuf::Message &from)
- void CopyFrom (const Link &from)
- void MergeFrom (const Link &from)
- void Clear ()
- bool IsInitialized () const
- int ByteSize () const
- bool MergePartialFromCodedStream (::google::protobuf::io::CodedInputStream ∗input)
- void SerializeWithCachedSizes (::google::protobuf::io::CodedOutputStream ∗output) const
- ::google::protobuf::uint8 ∗ SerializeWithCachedSizesToArray (::google::protobuf::uint8 ∗output) const
- int GetCachedSize () const
- ::google::protobuf::Metadata GetMetadata () const
- bool has_source () const
- void clear_source ()
- const ::std::string & source () const
- void set_source (const ::std::string &value)
- void set_source (const char ∗value)
- void set_source (const char ∗value, size_t size)
- inline::std::string ∗ mutable_source ()
- bool has_destination () const
- void clear_destination ()
- const ::std::string & destination () const

- void set_destination (const ::std::string &value)
- void set_destination (const char ∗value)
- void set_destination (const char ∗value, size_t size)
- inline::std::string ∗ mutable_destination ()
- bool has_weight () const
- void clear_weight ()
- double weight () const
- void set_weight (double value)

7.37.2 Static Public Member Functions

- static const ::google::protobuf::Descriptor ∗ descriptor ()
- static const Link & default_instance ()

7.37.3 Static Public Attributes

- static const int kSourceFieldNumber = 1
- static const int kDestinationFieldNumber = 2
- static const int kWeightFieldNumber = 3

7.37.4 Friends

- void protobuf_AddDesc_confcluster_2eproto ()
- void protobuf_AssignDesc_confcluster_2eproto ()
- void protobuf_ShutdownFile_confcluster_2eproto ()

7.37.5 Constructor & Destructor Documentation

Configurator::Link::Link ( )

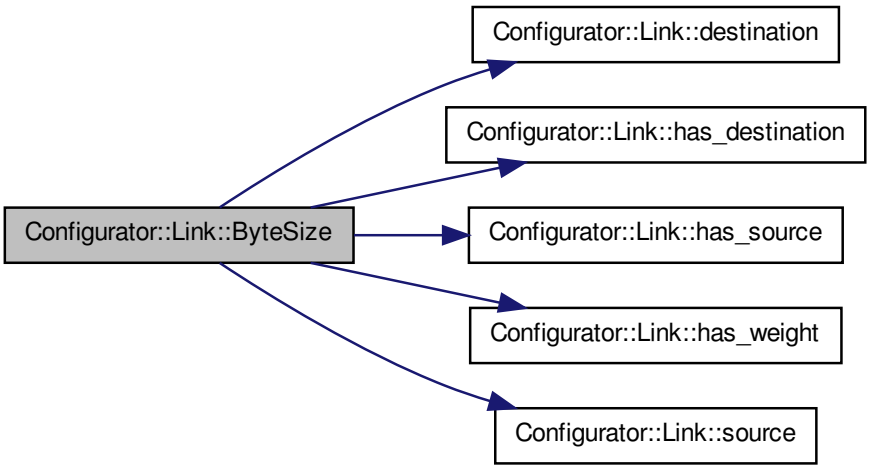
Configurator::Link::∼Link ( ) `[virtual]`

Configurator::Link::Link ([const Link &]from)

7.37.6 Member Function Documentation

int Configurator::Link::ByteSize ( ) const

Here is the call graph for this function:



void Configurator::Link::Clear ( )

void Configurator::Link::clear_destination ( ) [inline]

void Configurator::Link::clear_source ( ) [inline]

void Configurator::Link::clear_weight ( ) [inline]

void Configurator::Link::CopyFrom ([const Link &]from)

void Configurator::Link::CopyFrom ([const ::google::protobuf::Message &]from)

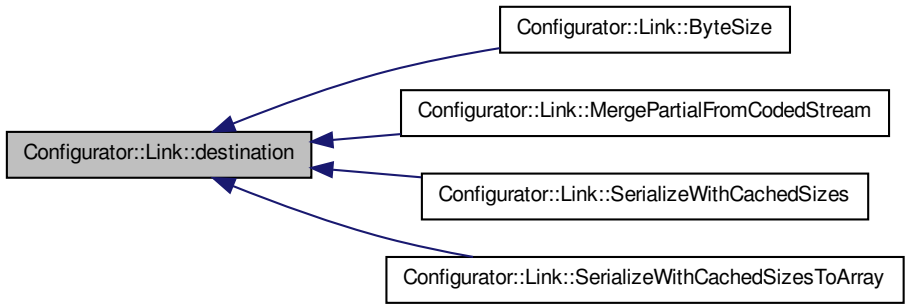
Here is the caller graph for this function:

const Link & Configurator::Link::default_instance ( ) [static]

const ::google::protobuf::Descriptor * Configurator::Link::descriptor ( ) [static]

const ::std::string & Configurator::Link::destination ( ) const [inline]

Here is the caller graph for this function:

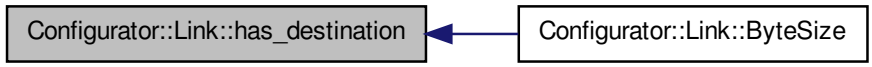


int Configurator::Link::GetCachedSize ( ) const [inline]

google::protobuf::Metadata Configurator::Link::GetMetadata ( ) const

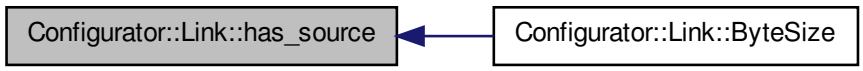
bool Configurator::Link::has_destination ( ) const [inline]

Here is the caller graph for this function:



bool Configurator::Link::has_source ( ) const [inline]

Here is the caller graph for this function:



bool Configurator::Link::has_weight ( ) const [inline]

Here is the caller graph for this function:

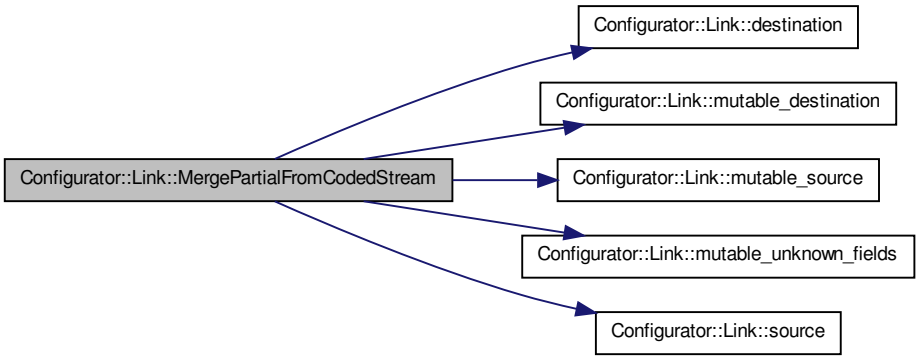
bool Configurator::Link::IsInitialized ( ) const

void Configurator::Link::MergeFrom ([const ::google::protobuf::Message &]from)

void Configurator::Link::MergeFrom ([const Link &]from)

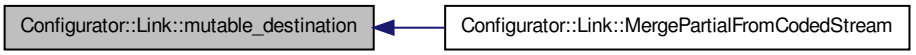
bool Configurator::Link::MergePartialFromCodedStream ([::google::protobuf::io::CodedInputStream ∗]input)

Here is the call graph for this function:



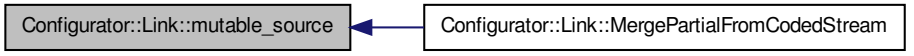
std::string ∗ Configurator::Link::mutable_destination ( ) `[inline]`

Here is the caller graph for this function:



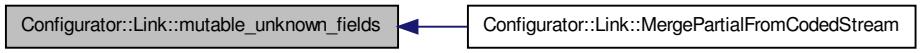
std::string ∗ Configurator::Link::mutable_source ( ) `[inline]`

Here is the caller graph for this function:



inline ::google::protobuf::UnknownFieldSet* Configurator::Link::mutable_unknown_fields ( ) [inline]

Here is the caller graph for this function:



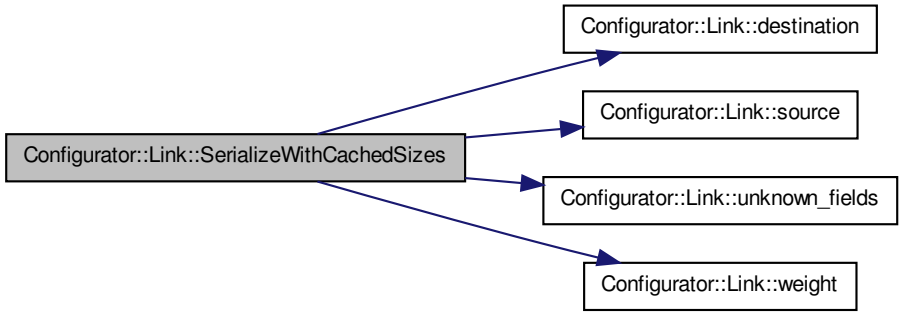
Link * Configurator::Link::New ( ) const

Link& Configurator::Link::operator= ([const Link &]from) [inline]

Here is the call graph for this function:

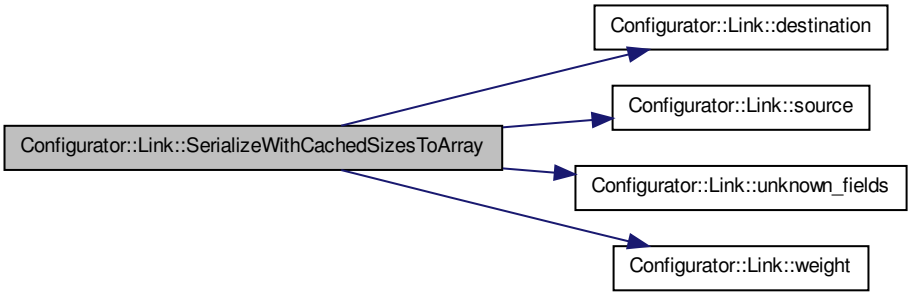
void Configurator::Link::SerializeWithCachedSizes ([::google::protobuf::io::CodedOutputStream *]output) const

Here is the call graph for this function:



google::protobuf::uint8 * Configurator::Link::SerializeWithCachedSizesToArray ([::google::protobuf::uint8 *]output) const

Here is the call graph for this function:



void Configurator::Link::set_destination ([const char ∗]value) [inline]

void Configurator::Link::set_destination ([const char ∗]value, size_t *size*) [inline]

void Configurator::Link::set_destination ([const ::std::string &]value) [inline]

void Configurator::Link::set_source ([const char ∗]value, size_t *size*) [inline]

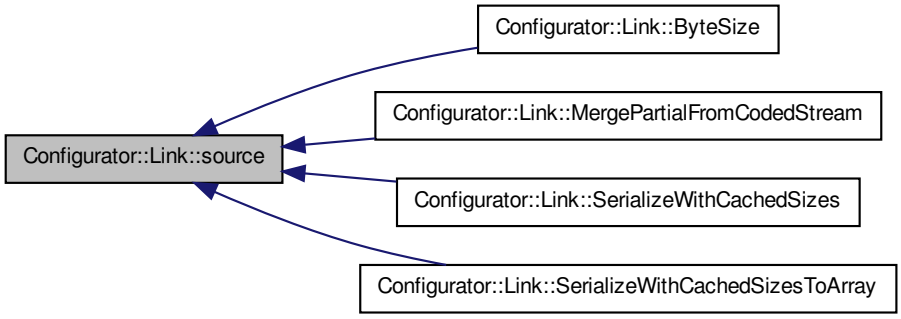
void Configurator::Link::set_source ([const ::std::string &]value) [inline]

void Configurator::Link::set_source ([const char ∗]value) [inline]

void Configurator::Link::set_weight ([double]value) [inline]

const ::std::string & Configurator::Link::source ( ) const [inline]

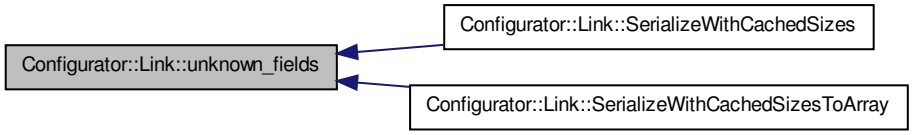
Here is the caller graph for this function:



void Configurator::Link::Swap ([Link ∗]other)

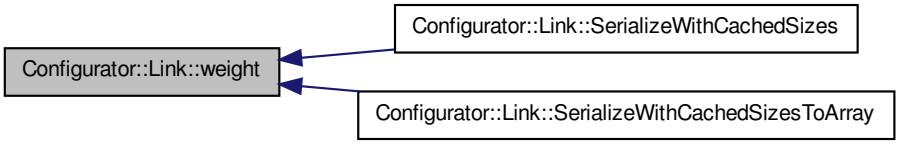
const ::google::protobuf::UnknownFieldSet& Configurator::Link::unknown_fields ( ) const [inline]

Here is the caller graph for this function:



double Configurator::Link::weight ( ) const `[inline]`

Here is the caller graph for this function:



7.37.7 Friends And Related Function Documentation

void protobuf_AddDesc_confcluster_2eproto ( ) [friend]

void protobuf_AssignDesc_confcluster_2eproto ( ) [friend]

void protobuf_ShutdownFile_confcluster_2eproto ( ) [friend]

7.37.8 Member Data Documentation

const int Configurator::Link::kDestinationFieldNumber = 2 [static]

const int Configurator::Link::kSourceFieldNumber = 1 [static]

const int Configurator::Link::kWeightFieldNumber = 3 [static]

The documentation for this class was generated from the following files:

- confcluster.pb.h
- confcluster.pb.cc

7.38 Reconfiguration::Link Class Reference

`#include <Graph.h>`

7.38.1 Public Member Functions

- Link ()

  Constructor of the Link class.

- Link (int, double)

  Constructor of the Machine class.

- virtual ∼Link ()

  virtual destructor of the class Link.

7.38.2 Public Attributes

- int iNeighbor
- double dLink

7.38.3 Detailed Description

Link between two nodes of the graph.

7.38.4 Constructor & Destructor Documentation

Reconfiguration::Link::Link ( )

Constructor of the Link class.

Returns

∗this

Creates an object Link.

Reconfiguration::Link::Link ([int]iDestination, double *dWeight*)

Constructor of the Machine class.

Parameters

in	*iDestination*	Neighbor identifier.
in	*dWeight*	Link weight.

Returns

∗this

Creates an object Link setting the neighbor and the weight variables.

Reconfiguration::Link::∼Link ( ) [virtual]

virtual destructor of the class Link.

Returns

void

Destroys the Link object.

7.38.5 Member Data Documentation

double Reconfiguration::Link::dLink

Weight of the link.

int Reconfiguration::Link::iNeighbor

Identifier of the neighbor vertex.

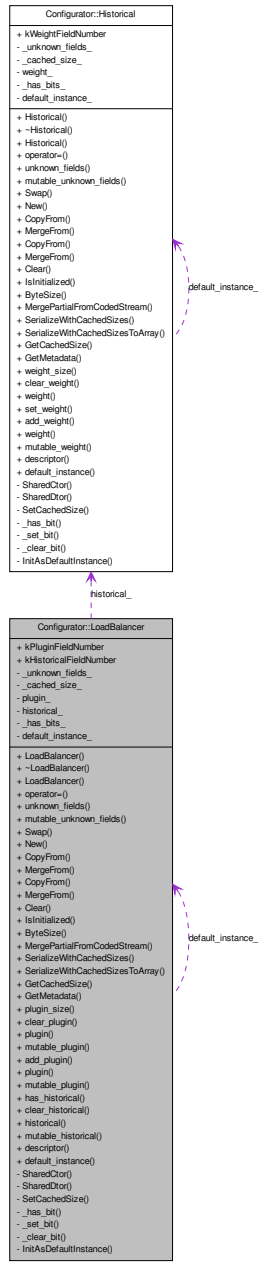
The documentation for this class was generated from the following files:

- Graph.h
- Graph.cpp

7.39 Configurator::LoadBalancer Class Reference

`#include <confloadbalancer.pb.h>`

Collaboration diagram for Configurator::LoadBalancer:



7.39.1 Public Member Functions

- LoadBalancer ()
- virtual ∼LoadBalancer ()
- LoadBalancer (const LoadBalancer &from)
- LoadBalancer & operator= (const LoadBalancer &from)
- const ::google::protobuf::UnknownFieldSet & unknown_fields () const
- inline::google::protobuf::UnknownFieldSet ∗ mutable_unknown_fields ()
- void Swap (LoadBalancer ∗other)
- LoadBalancer ∗ New () const
- void CopyFrom (const ::google::protobuf::Message &from)
- void MergeFrom (const ::google::protobuf::Message &from)
- void CopyFrom (const LoadBalancer &from)
- void MergeFrom (const LoadBalancer &from)
- void Clear ()
- bool IsInitialized () const
- int ByteSize () const
- bool MergePartialFromCodedStream (::google::protobuf::io::CodedInputStream ∗input)
- void SerializeWithCachedSizes (::google::protobuf::io::CodedOutputStream ∗output) const
- ::google::protobuf::uint8 ∗ SerializeWithCachedSizesToArray (::google::protobuf::uint8 ∗output) const
- int GetCachedSize () const
- ::google::protobuf::Metadata GetMetadata () const
- int plugin_size () const
- void clear_plugin ()
- const ::Configurator::Plugin & plugin (int index) const
- inline::Configurator::Plugin ∗ mutable_plugin (int index)
- inline::Configurator::Plugin ∗ add_plugin ()
- const ::google::protobuf::RepeatedPtrField< ::Configurator::Plugin > & plugin () const
- inline::google::protobuf::RepeatedPtrField< ::Configurator::Plugin > ∗ mutable_plugin ()
- bool has_historical () const
- void clear_historical ()
- const ::Configurator::Historical & historical () const

- inline::Configurator::Historical * mutable_historical ()

7.39.2 Static Public Member Functions

- static const ::google::protobuf::Descriptor * descriptor ()
- static const LoadBalancer & default_instance ()

7.39.3 Static Public Attributes

- static const int kPluginFieldNumber = 1
- static const int kHistoricalFieldNumber = 2

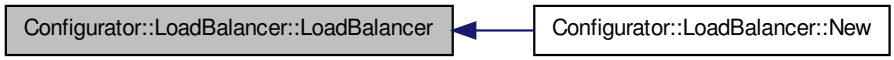
7.39.4 Friends

- void protobuf_AddDesc_confloadbalancer_2eproto ()
- void protobuf_AssignDesc_confloadbalancer_2eproto ()
- void protobuf_ShutdownFile_confloadbalancer_2eproto ()

7.39.5 Constructor & Destructor Documentation

Configurator::LoadBalancer::LoadBalancer ( )

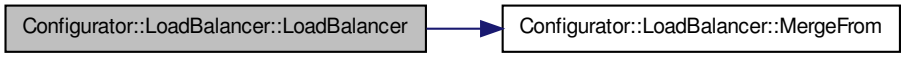
Here is the caller graph for this function:



Configurator::LoadBalancer::~LoadBalancer ( ) [virtual]

Configurator::LoadBalancer::LoadBalancer ([const LoadBalancer &]from)

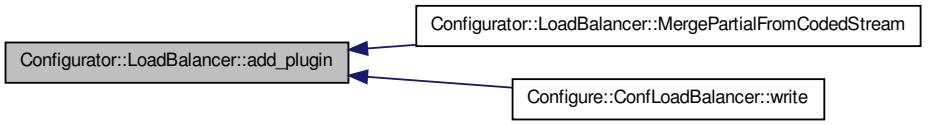
Here is the call graph for this function:



7.39.6 Member Function Documentation

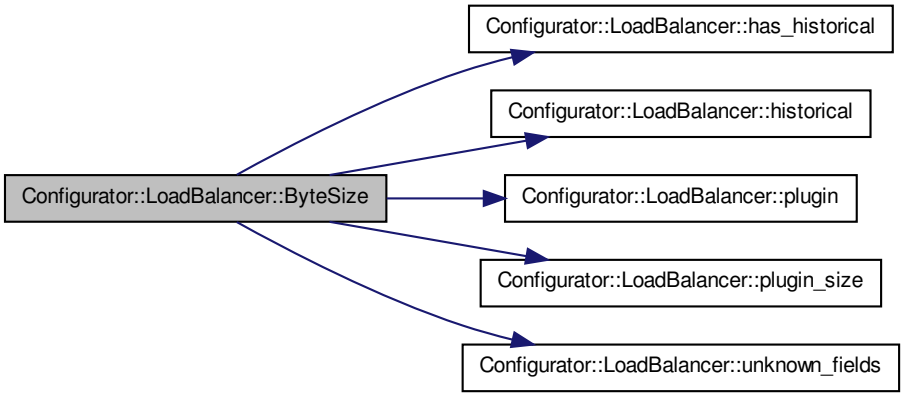
Configurator::Plugin * Configurator::LoadBalancer::add_plugin ( ) [inline]

Here is the caller graph for this function:



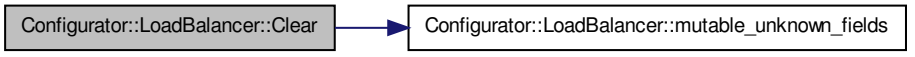
int Configurator::LoadBalancer::ByteSize ( ) const

Here is the call graph for this function:

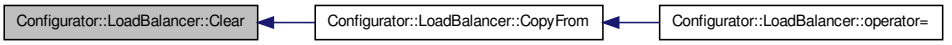


void Configurator::LoadBalancer::Clear ( )

Here is the call graph for this function:



Here is the caller graph for this function:

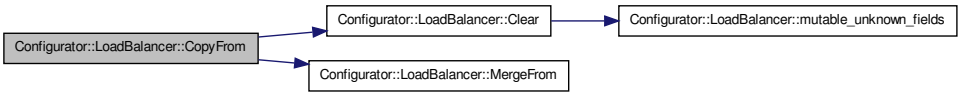


void Configurator::LoadBalancer::clear_historical ( ) [inline]

void Configurator::LoadBalancer::clear_plugin ( ) [inline]

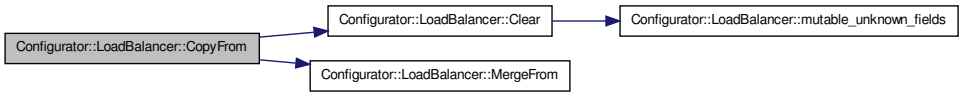
void Configurator::LoadBalancer::CopyFrom ([const LoadBalancer &]from)

Here is the call graph for this function:



void Configurator::LoadBalancer::CopyFrom ([const ::google::protobuf::Message &]from)

Here is the call graph for this function:



Here is the caller graph for this function:

const LoadBalancer & Configurator::LoadBalancer::default_instance ( ) [static]

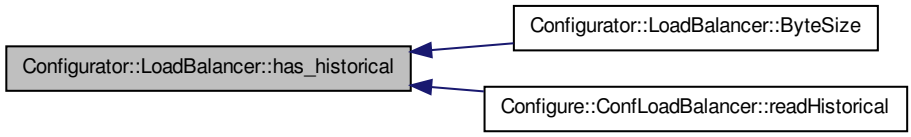
const ::google::protobuf::Descriptor * Configurator::LoadBalancer::descriptor ( ) [static]

int Configurator::LoadBalancer::GetCachedSize ( ) const [inline]

google::protobuf::Metadata Configurator::LoadBalancer::GetMetadata ( ) const

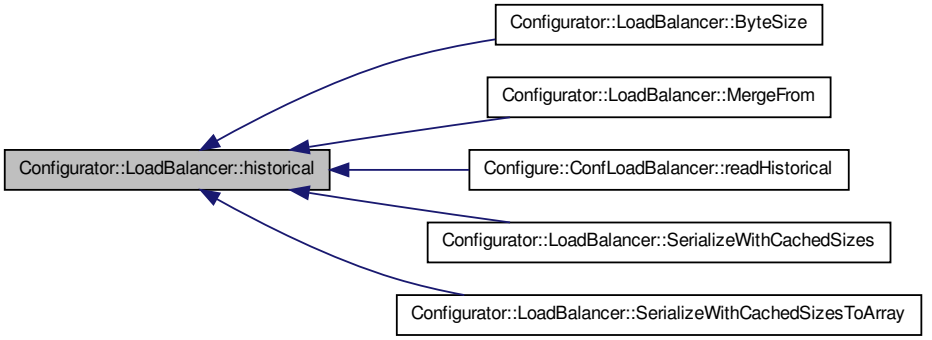
bool Configurator::LoadBalancer::has_historical ( ) const [inline]

Here is the caller graph for this function:



const ::Configurator::Historical & Configurator::LoadBalancer::historical ( ) const [inline]

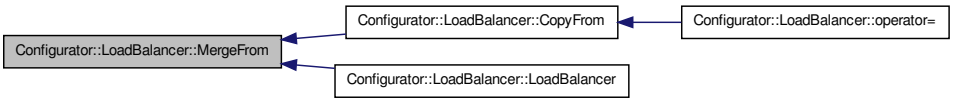
Here is the caller graph for this function:



bool Configurator::LoadBalancer::IsInitialized ( ) const

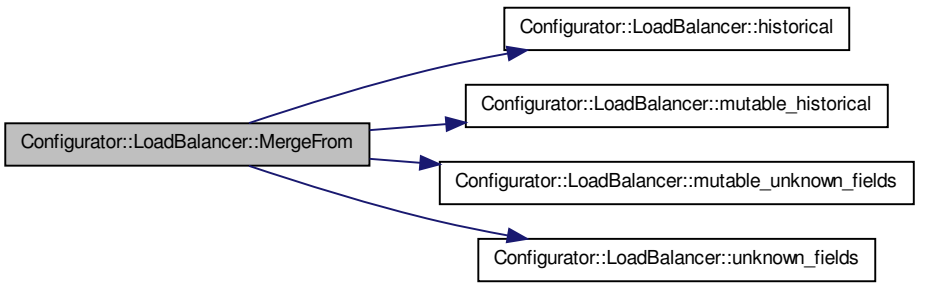
void Configurator::LoadBalancer::MergeFrom ([const ::google::protobuf::Message &]from)

Here is the caller graph for this function:



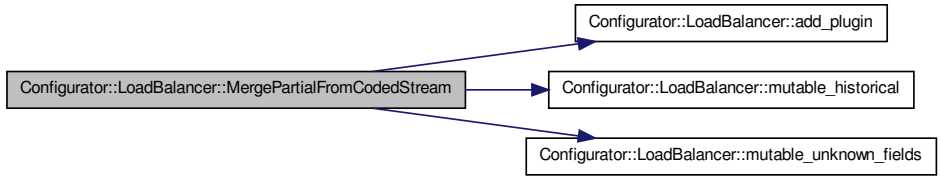
void Configurator::LoadBalancer::MergeFrom ([const LoadBalancer &]from)

Here is the call graph for this function:



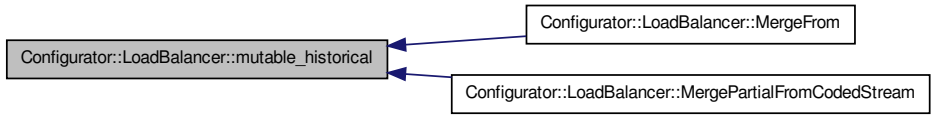
bool Configurator::LoadBalancer::MergePartialFromCodedStream ([::google::protobuf::io::CodedInputStream *]input)

Here is the call graph for this function:



Configurator::Historical * Configurator::LoadBalancer::mutable_historical ( ) [inline]

Here is the caller graph for this function:

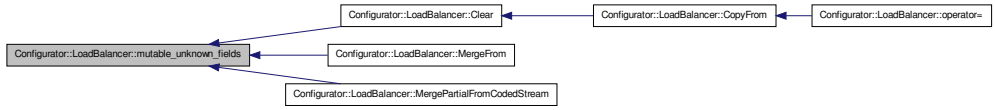


google::protobuf::RepeatedPtrField<::Configurator::Plugin > * Configurator::LoadBalancer::mutable_plugin ( ) [inline]

Configurator::Plugin * Configurator::LoadBalancer::mutable_plugin ([int]index) [inline]

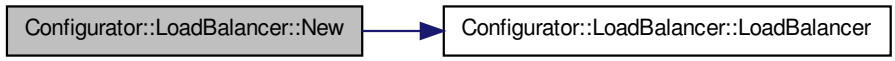
inline ::google::protobuf::UnknownFieldSet* Configurator::LoadBalancer::mutable_-unknown_fields ( ) [inline]

Here is the caller graph for this function:



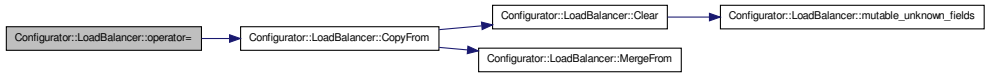
LoadBalancer * Configurator::LoadBalancer::New ( ) const

Here is the call graph for this function:



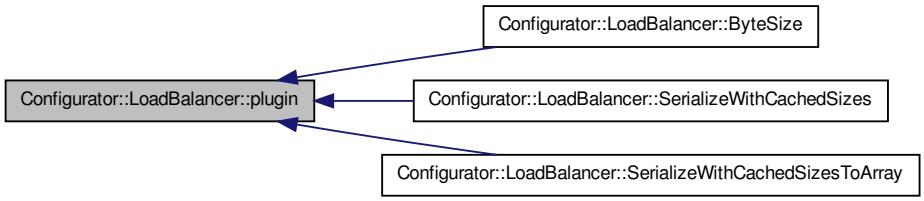
LoadBalancer& Configurator::LoadBalancer::operator= ([const LoadBalancer &]from) [inline]

Here is the call graph for this function:



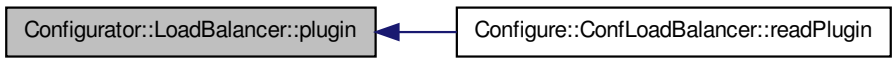
const ::google::protobuf::RepeatedPtrField<::Configurator::Plugin > & Configurator::LoadBalancer::plugin ( ) const [inline]

Here is the caller graph for this function:



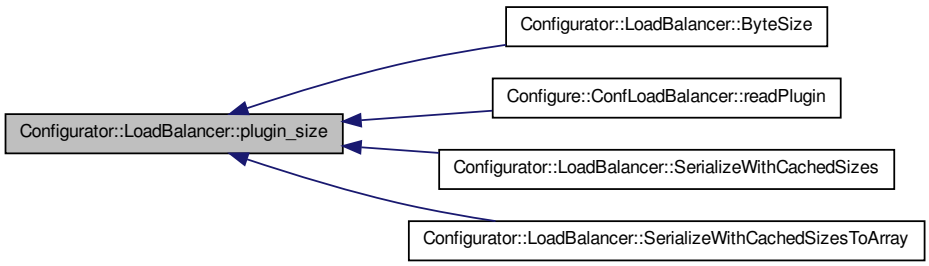
const ::Configurator::Plugin & Configurator::LoadBalancer::plugin ([int]index) const [inline]

Here is the caller graph for this function:



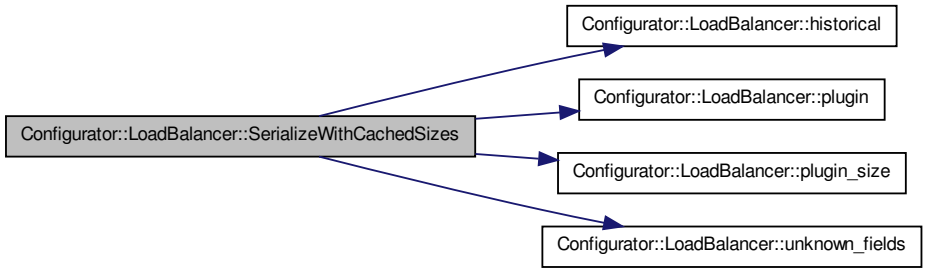
int Configurator::LoadBalancer::plugin_size ( ) const [inline]

Here is the caller graph for this function:



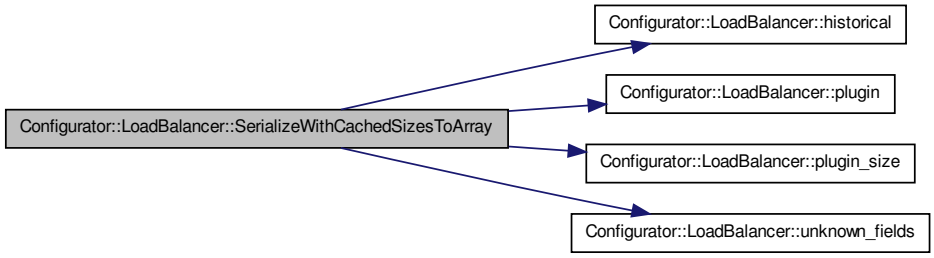
void Configurator::LoadBalancer::SerializeWithCachedSizes ([::google::protobuf::io::CodedOutputStream *]output) const

Here is the call graph for this function:



google::protobuf::uint8 * Configurator::LoadBalancer::SerializeWithCachedSizesToArray ([::google::protobuf::uint8 *]output) const

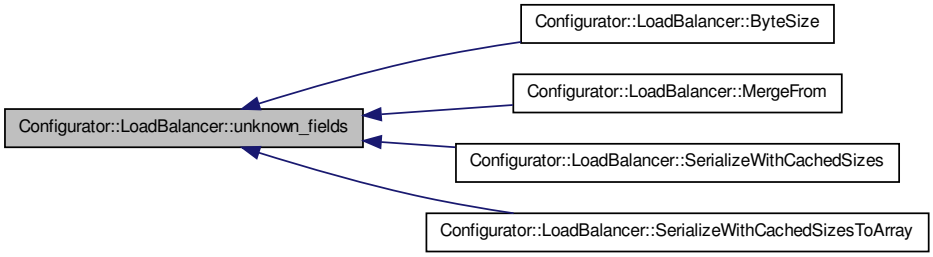
Here is the call graph for this function:



void Configurator::LoadBalancer::Swap ([LoadBalancer *]other)

const ::google::protobuf::UnknownFieldSet& Configurator::LoadBalancer::unknown_-fields ( ) const [inline]

Here is the caller graph for this function:



7.39.7 Friends And Related Function Documentation

void protobuf_AddDesc_confloadbalancer_2eproto ( ) [friend]

void protobuf_AssignDesc_confloadbalancer_2eproto ( ) [friend]

void protobuf_ShutdownFile_confloadbalancer_2eproto ( ) [friend]

7.39.8 Member Data Documentation

const int Configurator::LoadBalancer::kHistoricalFieldNumber = 2 [static]

const int Configurator::LoadBalancer::kPluginFieldNumber = 1 [static]

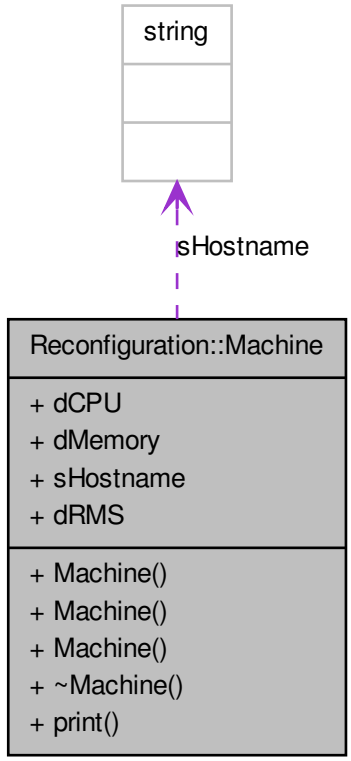
The documentation for this class was generated from the following files:

- confloadbalancer.pb.h
- confloadbalancer.pb.cc

7.40 Reconfiguration::Machine Class Reference

`#include <Graph.h>`

Collaboration diagram for Reconfiguration::Machine:



7.40.1 Public Member Functions

- Machine ()

  Constructor of the Machine class.

- Machine (const Machine &)

  Copy constructor of the Machine class.

- Machine (double, double, const std::string &, double)

  Constructor of the Machine class.

- virtual ~Machine ()

virtual destructor of the class Machine.

- void print ()

  prints the machine object.

7.40.2 Public Attributes

- double dCPU
- double dMemory
- std::string sHostname
- double dRMS

7.40.3 Detailed Description

Characteristics of the machine, part of the cluster.

7.40.4 Constructor & Destructor Documentation

Reconfiguration::Machine::Machine ( )

Constructor of the Machine class.

Returns

∗this

Creates an object Machine.

Reconfiguration::Machine::Machine ([const Machine &]machine)

Copy constructor of the Machine class.

Parameters

in	*machine*	Object to be copied.

Returns

∗this

Copies an object Machine.

Reconfiguration::Machine::Machine ([double]dCPU, double *dMemory,* const std::string & *sHostname,* double *dRMS*)

Constructor of the Machine class.

Parameters

in	*dCPU*	CPU frequency of the node.
in	*dMemory*	Free memory available of the node.
in	*sHostname*	Node's hostname.
in	*dRMS*	normalized RMS.

Returns

∗this

Creates an object Machine.

Reconfiguration::Machine::∼Machine ( ) [virtual]

virtual destructor of the class Machine.

Returns

void

Destroys the Machine object.

7.40.5 Member Function Documentation

void Reconfiguration::Machine::print ( )

prints the machine object.

Returns

void

Prints the machine object.

Here is the caller graph for this function:

7.40.6 Member Data Documentation

double Reconfiguration::Machine::dCPU

CPU Frequency (in MHz).

double Reconfiguration::Machine::dMemory

Free memory (in Mb).

double Reconfiguration::Machine::dRMS

Normalized value related to the nodes' characteristics and the plugins to be used for calculating the loadbalance.

std::string Reconfiguration::Machine::sHostname

Hostname of the machine.

The documentation for this class was generated from the following files:

- Graph.h
- Graph.cpp

7.41 Reconfiguration::MapElements Class Reference

`#include <Problem.h>`

7.41.1 Public Member Functions

- MapElements ()

  Constructor of the MapElements class.

- virtual ∼MapElements ()

  virtual destructor of the class MapElements.

- void insert (int, int)

  Inserts a new pair element-status in the map.

- void print ()

  prints the MapElement object.

7.41.2 Public Attributes

- std::map< int, int > elements

7.41.3 Detailed Description

Manages the access type of each graph vertex.

7.41.4 Constructor & Destructor Documentation

Reconfiguration::MapElements::MapElements ( )

Constructor of the MapElements class.

Returns

∗this

Creates an object MapElements.

Reconfiguration::MapElements::~MapElements ( ) [virtual]

virtual destructor of the class MapElements.

Returns

void

Destroys the MapElements object.

7.41.5 Member Function Documentation

void Reconfiguration::MapElements::insert ([int]iId, int *iStatus*)

Inserts a new pair element-status in the map.

Parameters

in	*iId*	Vertex identifier.
in	*iStatus*	Status of the vertex.

See also

UPDATE

Returns

void

Inserts a new pair element-status in the map. Before the insertion the element is searched in the map. If it exists, it is only inserted if the status was not RW. If the status was RW, the insertion is ignored.

Parameters

in	*iId*	Vertex identifier.
in	*iStatus*	Status of the vertex (

See also

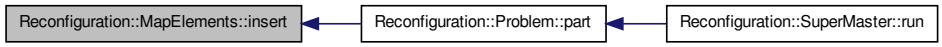
UPDATE).

Returns

void

Inserts a new pair element-status in the map. Before the insertion the element is searched in the map. If it exists, it is only inserted if the status was not RW. If the status was RW, the insertion is ignored.

Here is the caller graph for this function:



void Reconfiguration::MapElements::print ( )

prints the MapElement object.

Returns

void

Prints the MapElement object.

7.41.6 Member Data Documentation

std::map< int, int > Reconfiguration::MapElements::elements

Pairs vertex identifier vs vertex access.

See also

UPDATE

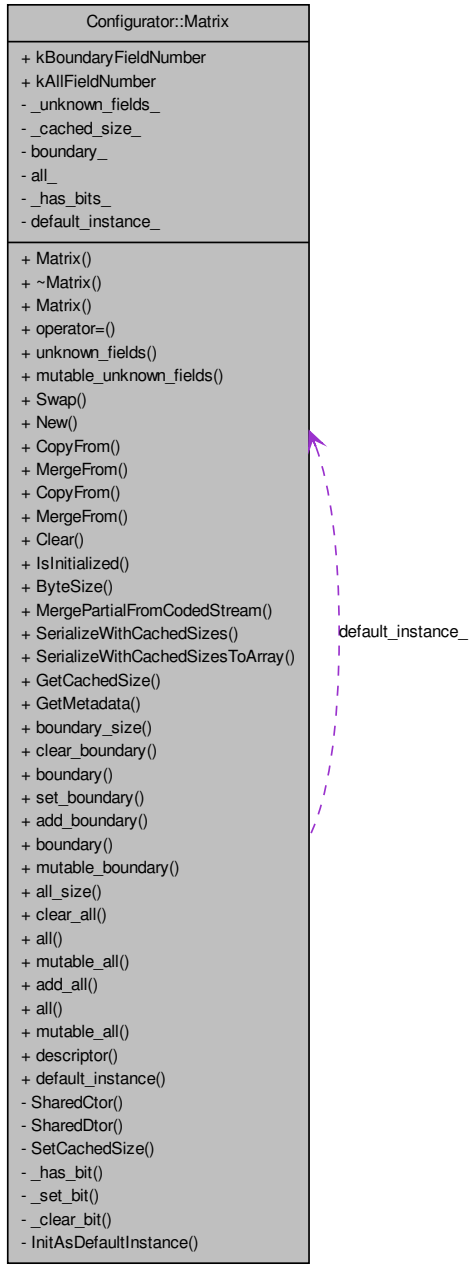
The documentation for this class was generated from the following files:

- Problem.h
- Problem.cpp

7.42 Configurator::Matrix Class Reference

`#include <confproblem.pb.h>`

Collaboration diagram for Configurator::Matrix:



7.42.1 Public Member Functions

- Matrix ()
- virtual ∼Matrix ()
- Matrix (const Matrix &from)
- Matrix & operator= (const Matrix &from)
- const ::google::protobuf::UnknownFieldSet & unknown_fields () const
- inline::google::protobuf::UnknownFieldSet ∗ mutable_unknown_fields ()
- void Swap (Matrix ∗other)
- Matrix ∗ New () const
- void CopyFrom (const ::google::protobuf::Message &from)
- void MergeFrom (const ::google::protobuf::Message &from)
- void CopyFrom (const Matrix &from)
- void MergeFrom (const Matrix &from)
- void Clear ()
- bool IsInitialized () const
- int ByteSize () const
- bool MergePartialFromCodedStream (::google::protobuf::io::CodedInputStream ∗input)
- void SerializeWithCachedSizes (::google::protobuf::io::CodedOutputStream ∗output) const
- ::google::protobuf::uint8 ∗ SerializeWithCachedSizesToArray (::google::protobuf::uint8 ∗output) const
- int GetCachedSize () const
- ::google::protobuf::Metadata GetMetadata () const
- int boundary_size () const
- void clear_boundary ()
- bool boundary (int index) const
- void set_boundary (int index, bool value)
- void add_boundary (bool value)
- const ::google::protobuf::RepeatedField< bool > & boundary () const
- inline::google::protobuf::RepeatedField< bool > ∗ mutable_boundary ()
- int all_size () const
- void clear_all ()
- const ::Configurator::Connection & all (int index) const

- inline::Configurator::Connection * mutable_all (int index)
- inline::Configurator::Connection * add_all ()
- const ::google::protobuf::RepeatedPtrField< ::Configurator::Connection > & all () const
- inline::google::protobuf::RepeatedPtrField< ::Configurator::Connection > * mutable_-all ()

7.42.2 Static Public Member Functions

- static const ::google::protobuf::Descriptor * descriptor ()
- static const Matrix & default_instance ()

7.42.3 Static Public Attributes

- static const int kBoundaryFieldNumber = 1
- static const int kAllFieldNumber = 2

7.42.4 Friends

- void protobuf_AddDesc_confproblem_2eproto ()
- void protobuf_AssignDesc_confproblem_2eproto ()
- void protobuf_ShutdownFile_confproblem_2eproto ()

7.42.5 Constructor & Destructor Documentation

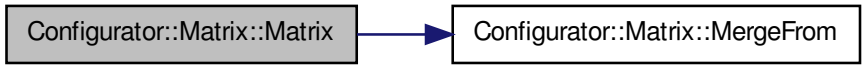
Configurator::Matrix::Matrix ( )

Here is the caller graph for this function:

Configurator::Matrix::~Matrix ( ) [virtual]

Configurator::Matrix::Matrix ([const Matrix &]from)

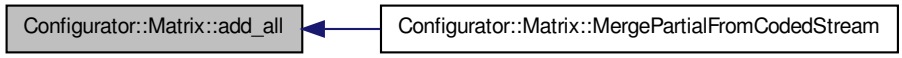
Here is the call graph for this function:



7.42.6 Member Function Documentation

Configurator::Connection * Configurator::Matrix::add_all ( ) [inline]

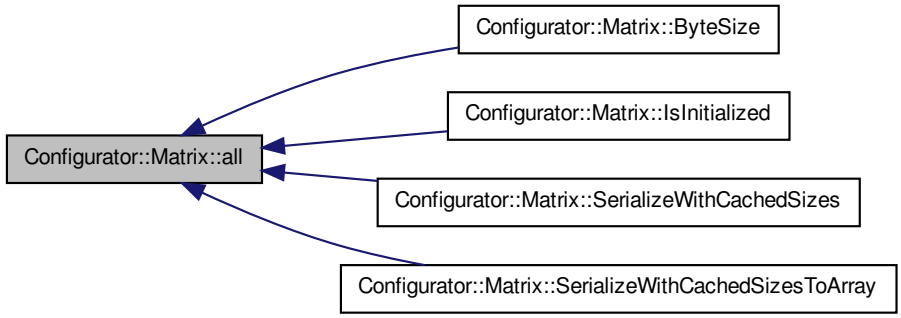
Here is the caller graph for this function:



void Configurator::Matrix::add_boundary ([bool]value) [inline]

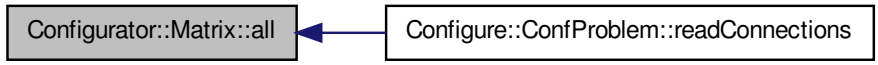
const ::google::protobuf::RepeatedPtrField<::Configurator::Connection > & Configurator::Matrix::all ( ) const [inline]

Here is the caller graph for this function:



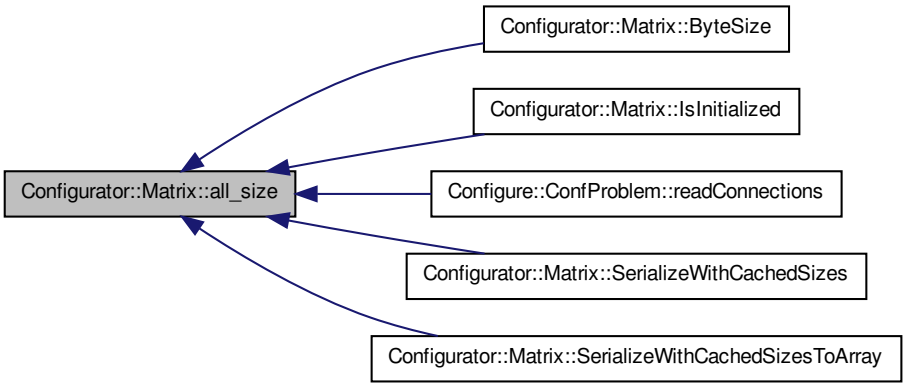
const ::Configurator::Connection & Configurator::Matrix::all ([int]index) const [inline]

Here is the caller graph for this function:



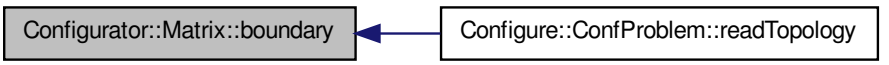
int Configurator::Matrix::all_size ( ) const `[inline]`

Here is the caller graph for this function:



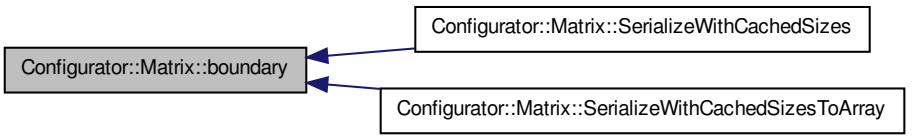
bool Configurator::Matrix::boundary ([int]index) const `[inline]`

Here is the caller graph for this function:



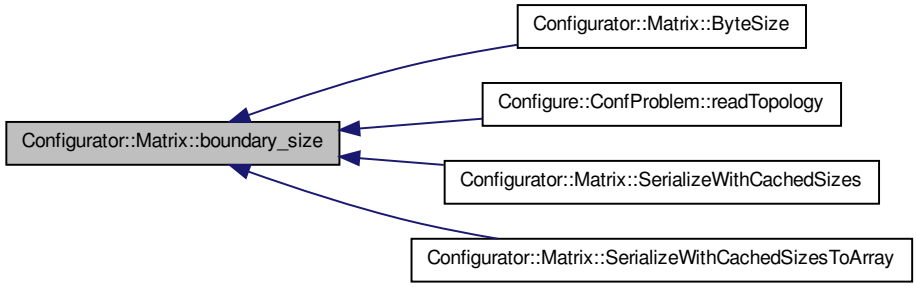
const ::google::protobuf::RepeatedField< bool > & Configurator::Matrix::boundary ( ) const `[inline]`

Here is the caller graph for this function:



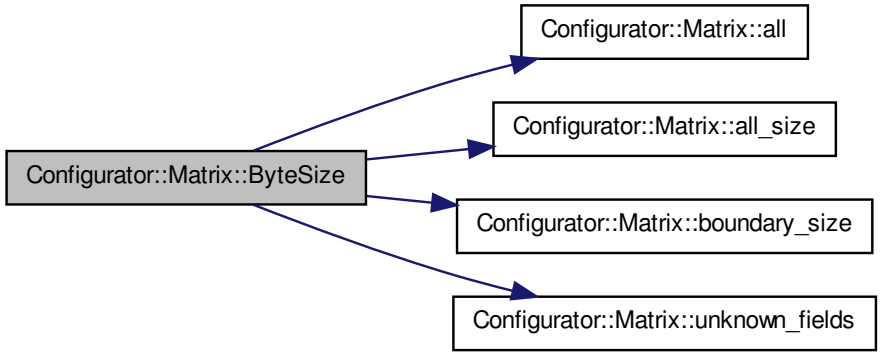
int Configurator::Matrix::boundary_size ( ) const `[inline]`

Here is the caller graph for this function:



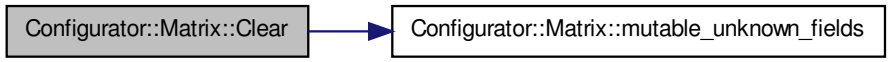
int Configurator::Matrix::ByteSize ( ) const

Here is the call graph for this function:

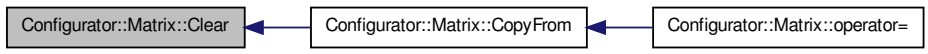


void Configurator::Matrix::Clear ( )

Here is the call graph for this function:



Here is the caller graph for this function:

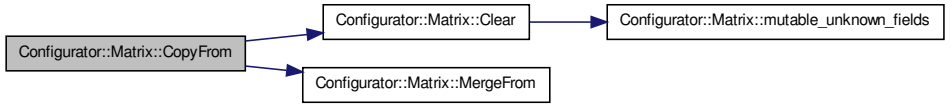


void Configurator::Matrix::clear_all ( ) [inline]

void Configurator::Matrix::clear_boundary ( ) [inline]

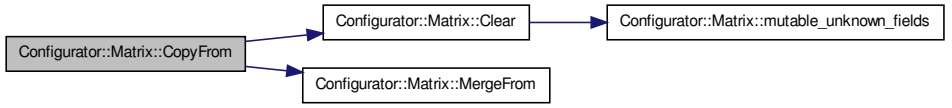
void Configurator::Matrix::CopyFrom ([const Matrix &]from)

Here is the call graph for this function:



void Configurator::Matrix::CopyFrom ([const ::google::protobuf::Message &]from)

Here is the call graph for this function:



Here is the caller graph for this function:

const Matrix & Configurator::Matrix::default_instance ( ) [static]

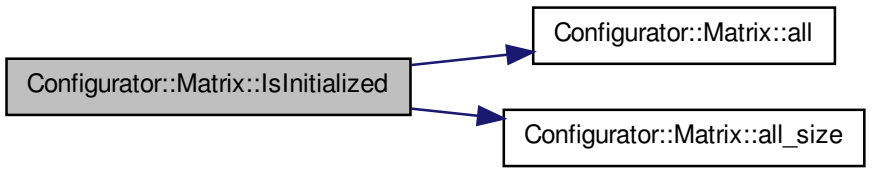
const ::google::protobuf::Descriptor * Configurator::Matrix::descriptor ( ) [static]

int Configurator::Matrix::GetCachedSize ( ) const [inline]

google::protobuf::Metadata Configurator::Matrix::GetMetadata ( ) const

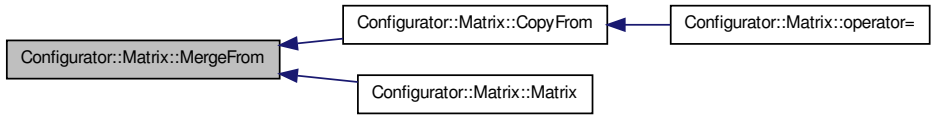
bool Configurator::Matrix::IsInitialized ( ) const

Here is the call graph for this function:



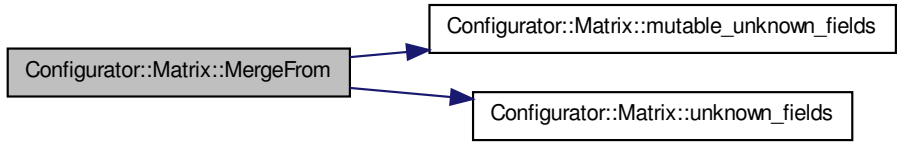
void Configurator::Matrix::MergeFrom ([const ::google::protobuf::Message &]from)

Here is the caller graph for this function:



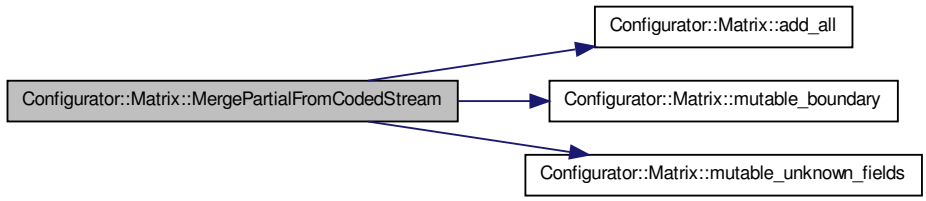
void Configurator::Matrix::MergeFrom ([const Matrix &]from)

Here is the call graph for this function:



bool Configurator::Matrix::MergePartialFromCodedStream ([::google::protobuf::io::CodedInputStream *]input)

Here is the call graph for this function:

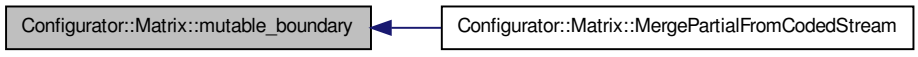


google::protobuf::RepeatedPtrField<::Configurator::Connection > * Configurator::Matrix::mutable_all ( ) [inline]

Configurator::Connection * Configurator::Matrix::mutable_all ([int]index) [inline]

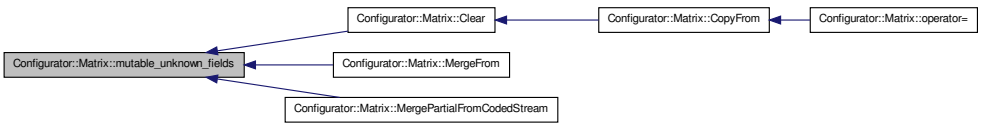
google::protobuf::RepeatedField< bool > * Configurator::Matrix::mutable_boundary ( ) [inline]

Here is the caller graph for this function:



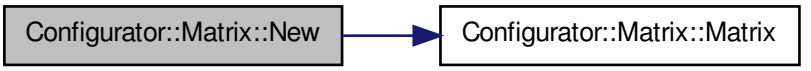
inline ::google::protobuf::UnknownFieldSet* Configurator::Matrix::mutable_-unknown_fields ( ) [inline]

Here is the caller graph for this function:



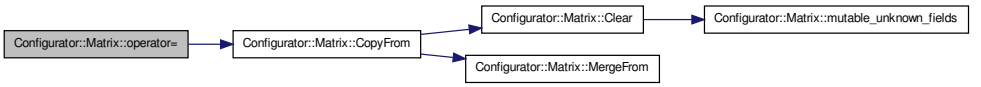
Matrix * Configurator::Matrix::New ( ) const

Here is the call graph for this function:



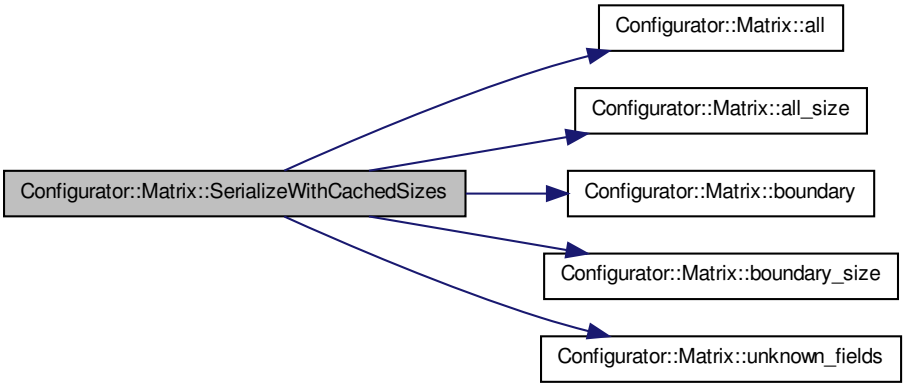
Matrix& Configurator::Matrix::operator= ([const Matrix &]from) `[inline]`

Here is the call graph for this function:



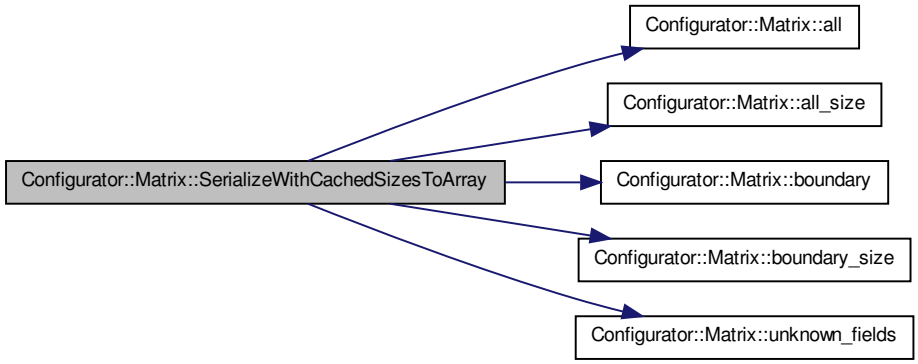
void Configurator::Matrix::SerializeWithCachedSizes ([::google::protobuf::io::CodedOutputStream *]output) const

Here is the call graph for this function:



google::protobuf::uint8 * Configurator::Matrix::SerializeWithCachedSizesToArray ([::google::protobuf::uint8 *]output) const

Here is the call graph for this function:

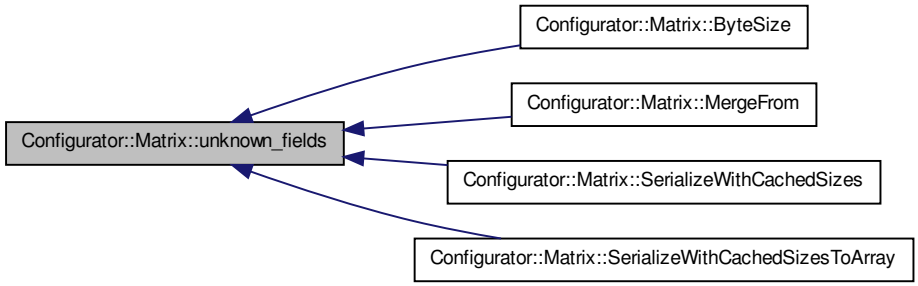


void Configurator::Matrix::set_boundary ([int]index, bool *value*) `[inline]`

void Configurator::Matrix::Swap ([Matrix *]other)

const ::google::protobuf::UnknownFieldSet& Configurator::Matrix::unknown_fields ( ) const `[inline]`

Here is the caller graph for this function:



7.42.7 Friends And Related Function Documentation

void protobuf_AddDesc_confproblem_2eproto ( ) `[friend]`

void protobuf_AssignDesc_confproblem_2eproto ( ) `[friend]`

void protobuf_ShutdownFile_confproblem_2eproto ( ) `[friend]`

7.42.8 Member Data Documentation

const int Configurator::Matrix::kAllFieldNumber = 2 `[static]`

const int Configurator::Matrix::kBoundaryFieldNumber = 1 `[static]`

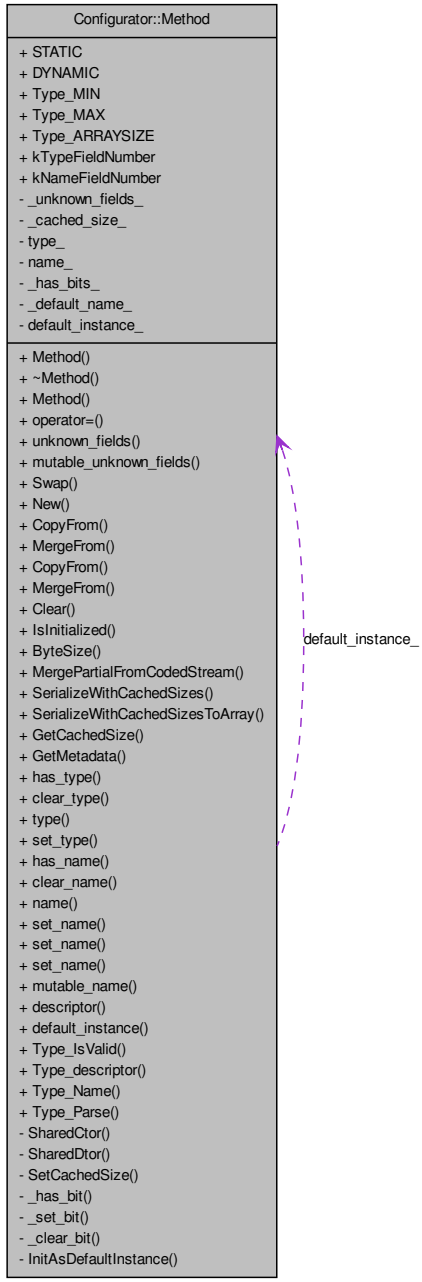
The documentation for this class was generated from the following files:

- confproblem.pb.h
- confproblem.pb.cc

7.43 Configurator::Method Class Reference

`#include <confpartitioner.pb.h>`

Collaboration diagram for Configurator::Method:



7.43.1 Public Types

- typedef Method_Type Type

7.43.2 Public Member Functions

- Method ()
- virtual ~Method ()
- Method (const Method &from)
- Method & operator= (const Method &from)
- const ::google::protobuf::UnknownFieldSet & unknown_fields () const
- inline::google::protobuf::UnknownFieldSet * mutable_unknown_fields ()
- void Swap (Method *other)
- Method * New () const
- void CopyFrom (const ::google::protobuf::Message &from)
- void MergeFrom (const ::google::protobuf::Message &from)
- void CopyFrom (const Method &from)
- void MergeFrom (const Method &from)
- void Clear ()
- bool IsInitialized () const
- int ByteSize () const
- bool MergePartialFromCodedStream (::google::protobuf::io::CodedInputStream *input)
- void SerializeWithCachedSizes (::google::protobuf::io::CodedOutputStream *output) const
- ::google::protobuf::uint8 * SerializeWithCachedSizesToArray (::google::protobuf::uint8 *output) const
- int GetCachedSize () const
- ::google::protobuf::Metadata GetMetadata () const
- bool has_type () const
- void clear_type ()
- inline::Configurator::Method_Type type () const
- void set_type (::Configurator::Method_Type value)
- bool has_name () const
- void clear_name ()

- const ::std::string & name () const
- void set_name (const ::std::string &value)
- void set_name (const char ∗value)
- void set_name (const char ∗value, size_t size)
- inline::std::string ∗ mutable_name ()

7.43.3 Static Public Member Functions

- static const ::google::protobuf::Descriptor ∗ descriptor ()
- static const Method & default_instance ()
- static bool Type_IsValid (int value)
- static const ::google::protobuf::EnumDescriptor ∗ Type_descriptor ()
- static const ::std::string & Type_Name (Type value)
- static bool Type_Parse (const ::std::string &name, Type ∗value)

7.43.4 Static Public Attributes

- static const Type STATIC = Method_Type_STATIC
- static const Type DYNAMIC = Method_Type_DYNAMIC
- static const Type Type_MIN
- static const Type Type_MAX
- static const int Type_ARRAYSIZE
- static const int kTypeFieldNumber = 1
- static const int kNameFieldNumber = 2

7.43.5 Friends

- void protobuf_AddDesc_confpartitioner_2eproto ()
- void protobuf_AssignDesc_confpartitioner_2eproto ()
- void protobuf_ShutdownFile_confpartitioner_2eproto ()

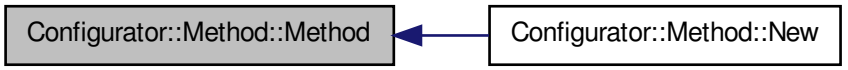
7.43.6 Member Typedef Documentation

typedef Method_Type Configurator::Method::Type

7.43.7 Constructor & Destructor Documentation

Configurator::Method::Method ( )

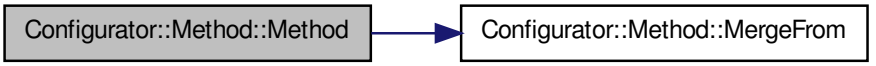
Here is the caller graph for this function:



Configurator::Method::~Method ( ) [virtual]

Configurator::Method::Method ([const Method &]from)

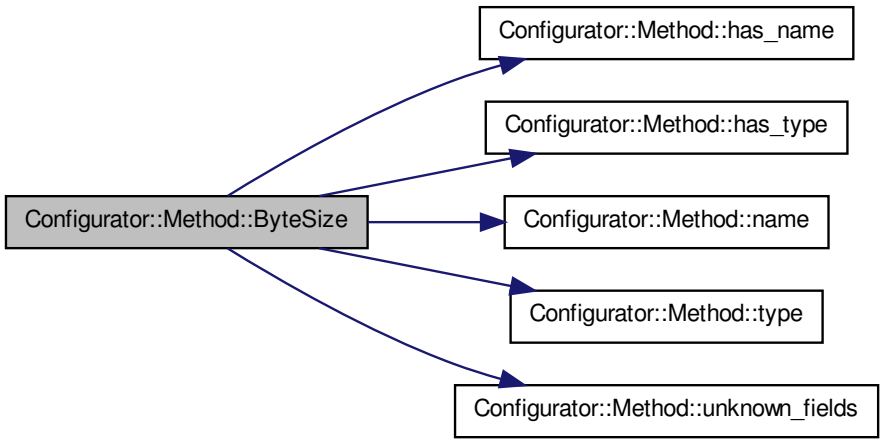
Here is the call graph for this function:



7.43.8 Member Function Documentation

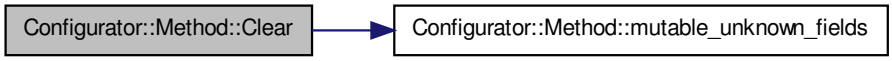
int Configurator::Method::ByteSize ( ) const

Here is the call graph for this function:

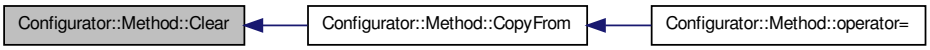


void Configurator::Method::Clear ( )

Here is the call graph for this function:



Here is the caller graph for this function:

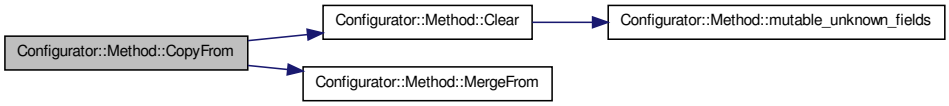


void Configurator::Method::clear_name ( ) [inline]

void Configurator::Method::clear_type ( ) [inline]

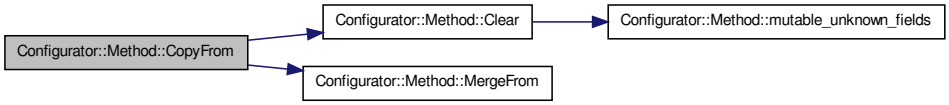
void Configurator::Method::CopyFrom ([const Method &]from)

Here is the call graph for this function:



void Configurator::Method::CopyFrom ([const ::google::protobuf::Message &]from)

Here is the call graph for this function:



Here is the caller graph for this function:

const Method & Configurator::Method::default_instance ( ) [static]

const ::google::protobuf::Descriptor * Configurator::Method::descriptor ( ) [static]

int Configurator::Method::GetCachedSize ( ) const [inline]

google::protobuf::Metadata Configurator::Method::GetMetadata ( ) const

bool Configurator::Method::has_name ( ) const [inline]

Here is the caller graph for this function:

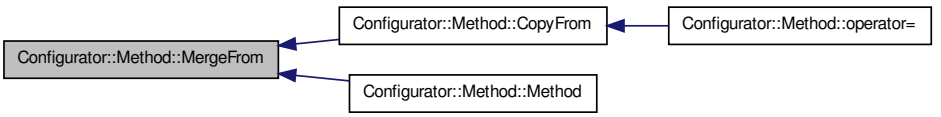
bool Configurator::Method::has_type ( ) const [inline]

Here is the caller graph for this function:

bool Configurator::Method::IsInitialized ( ) const

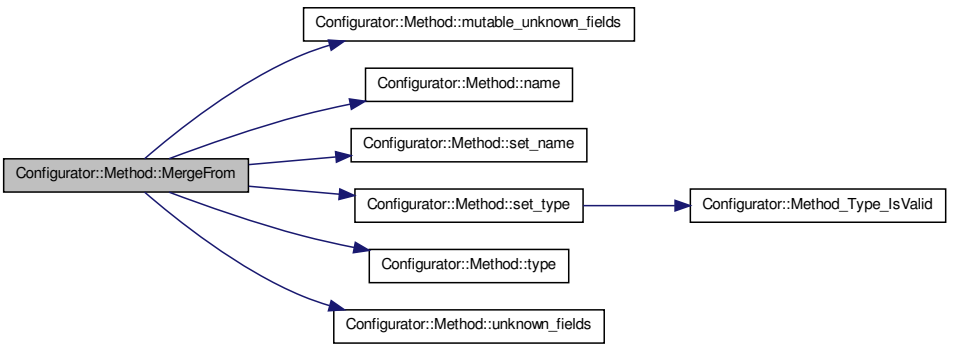
void Configurator::Method::MergeFrom ([const ::google::protobuf::Message &]from)

Here is the caller graph for this function:



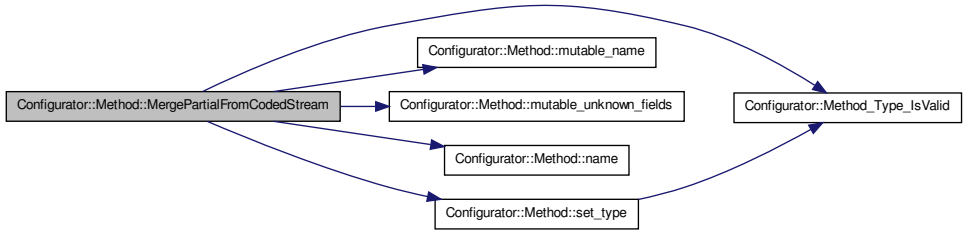
void Configurator::Method::MergeFrom ([const Method &]from)

Here is the call graph for this function:



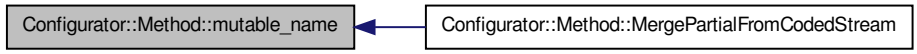
bool Configurator::Method::MergePartialFromCodedStream ([::google::protobuf::io::CodedInputStream *]input)

Here is the call graph for this function:



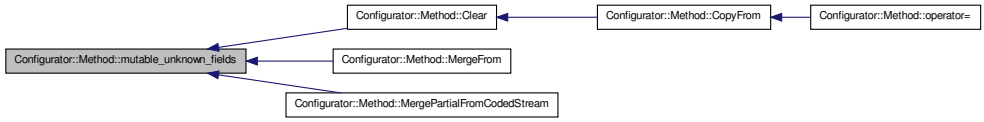
std::string * Configurator::Method::mutable_name ( ) [inline]

Here is the caller graph for this function:



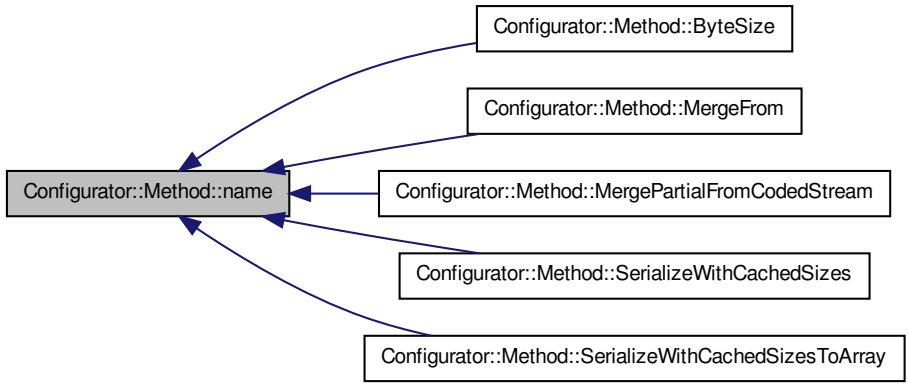
inline ::google::protobuf::UnknownFieldSet* Configurator::Method::mutable_-unknown_fields ( ) [inline]

Here is the caller graph for this function:



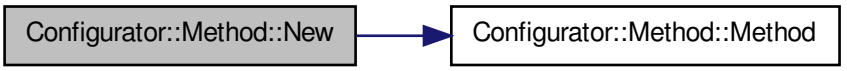
const ::std::string & Configurator::Method::name ( ) const [inline]

Here is the caller graph for this function:



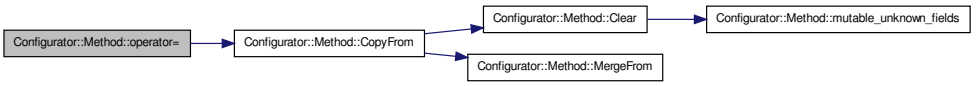
Method * Configurator::Method::New ( ) const

Here is the call graph for this function:



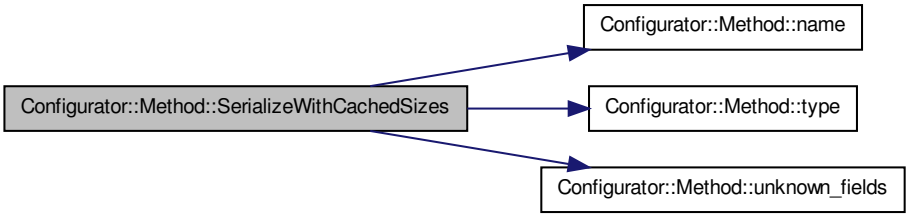
Method& Configurator::Method::operator= ([const Method &]from) [inline]

Here is the call graph for this function:



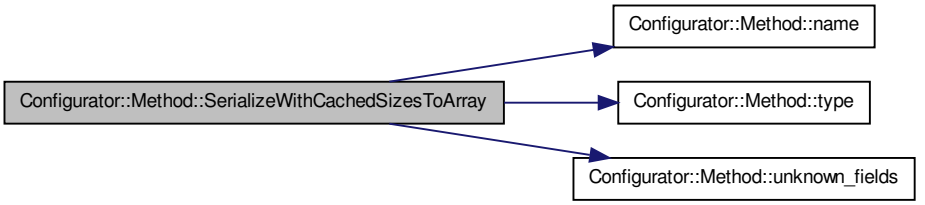
void Configurator::Method::SerializeWithCachedSizes ([::google::protobuf::io::CodedOutputStream *]output) const

Here is the call graph for this function:



google::protobuf::uint8 * Configurator::Method::SerializeWithCachedSizesToArray ([::google::protobuf::uint8 *]output) const

Here is the call graph for this function:



void Configurator::Method::set_name ([const char *]value) [inline]

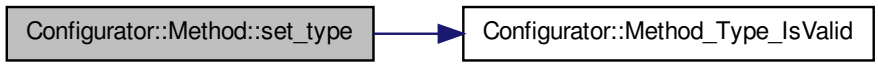
void Configurator::Method::set_name ([const ::std::string &]value) [inline]

Here is the caller graph for this function:

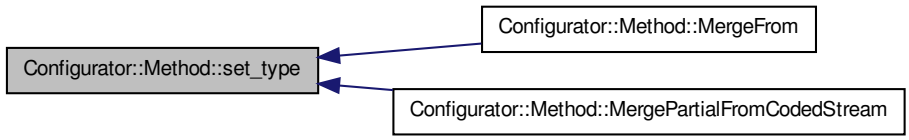
void Configurator::Method::set_name ([const char *]value, size_t *size*) `[inline]`

void Configurator::Method::set_type ([::Configurator::Method_Type]value) `[inline]`

Here is the call graph for this function:



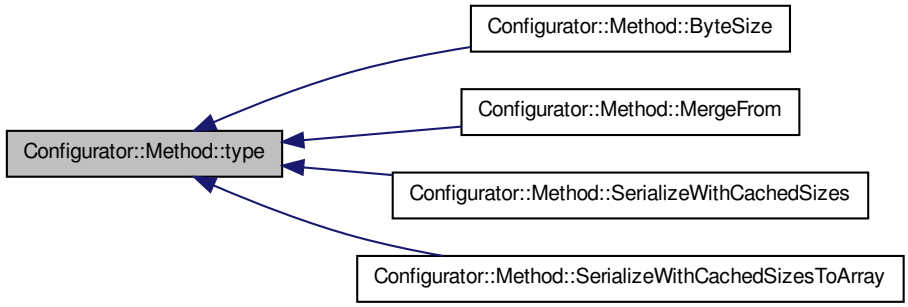
Here is the caller graph for this function:



void Configurator::Method::Swap ([Method *]other)

Configurator::Method_Type Configurator::Method::type ( ) const `[inline]`

Here is the caller graph for this function:



static const ::google::protobuf::EnumDescriptor* Configurator::Method::Type_descriptor ( ) `[inline, static]`

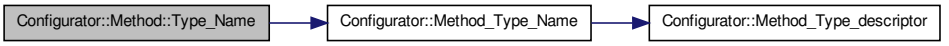
Here is the call graph for this function:

static bool Configurator::Method::Type_IsValid ([int]value) `[inline, static]`

Here is the call graph for this function:

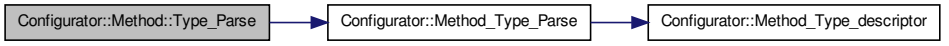
static const ::std::string& Configurator::Method::Type_Name ([Type]value) `[inline, static]`

Here is the call graph for this function:



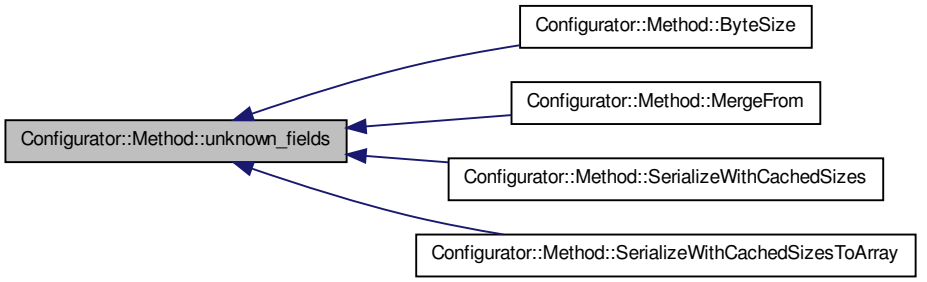
static bool Configurator::Method::Type_Parse ([const ::std::string &]name, Type * *value*) `[inline, static]`

Here is the call graph for this function:



const ::google::protobuf::UnknownFieldSet& Configurator::Method::unknown_fields ( ) const `[inline]`

Here is the caller graph for this function:



7.43.9 Friends And Related Function Documentation

void protobuf_AddDesc_confpartitioner_2eproto ( ) [friend]

void protobuf_AssignDesc_confpartitioner_2eproto ( ) [friend]

void protobuf_ShutdownFile_confpartitioner_2eproto ( ) [friend]

7.43.10 Member Data Documentation

const Method_Type Configurator::Method::DYNAMIC = Method_Type_DYNAMIC [static]

const int Configurator::Method::kNameFieldNumber = 2 [static]

const int Configurator::Method::kTypeFieldNumber = 1 [static]

const Method_Type Configurator::Method::STATIC = Method_Type_STATIC [static]

const int Configurator::Method::Type_ARRAYSIZE [static]

Initial value:

```
Method_Type_Type_ARRAYSIZE
```

const Method_Type Configurator::Method::Type_MAX [static]

Initial value:

```
Method_Type_Type_MAX
```

const Method_Type Configurator::Method::Type_MIN [static]

Initial value:

```
Method_Type_Type_MIN
```

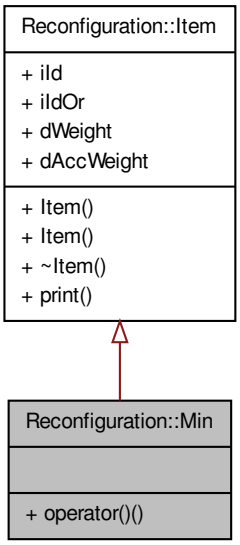
The documentation for this class was generated from the following files:

- confpartitioner.pb.h
- confpartitioner.pb.cc

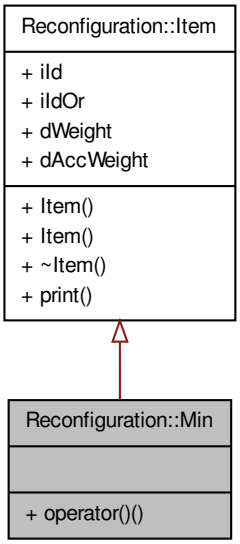
7.44 Reconfiguration::Min Class Reference

`#include <Min.h>`

Inheritance diagram for Reconfiguration::Min:



Collaboration diagram for Reconfiguration::Min:



7.44.1 Public Member Functions

- bool operator() (Item i, Item j)

  Compares two elements.

7.44.2 Detailed Description

Needed to sort the elements of the queue for the IGrouping interface.

7.44.3 Member Function Documentation

bool Reconfiguration::Min::operator() ([Item]i, Item *j*) `[inline]`

Compares two elements.

Parameters

in	*i*	First item.
in	*j*	Second item.

Returns

bool

Returns false if the second element has a biggest or equal value at the dAccWeight field or 1 Otherwise.

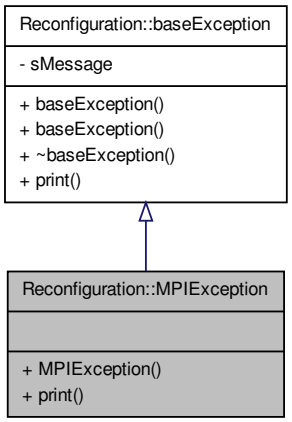
The documentation for this class was generated from the following file:

- Min.h

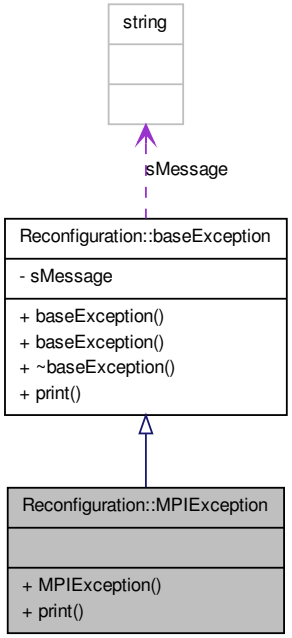
7.45 Reconfiguration::MPIException Class Reference

`#include <Exception.h>`

Inheritance diagram for Reconfiguration::MPIException:



Collaboration diagram for Reconfiguration::MPIException:



7.45.1 Public Member Functions

- MPIException (int)

  Constructor of the MPIException class.

- void print ()

  Prints the MPIException exception.

7.45.2 Detailed Description

Exception thrown if a MPI error occurs.

7.45.3 Constructor & Destructor Documentation

Reconfiguration::MPIException::MPIException ([int]iError)

Constructor of the MPIException class.

Parameters

in	*iError*	MPI error code.

Returns

∗this

Creates an object MPIException calling the baseException constructor and sets the message of the exception.

Here is the call graph for this function:

7.45.4 Member Function Documentation

void Reconfiguration::MPIException::print ( )

Prints the MPIException exception.

Returns

void

Prints the exception by printing its message.

Reimplemented from Reconfiguration::baseException.

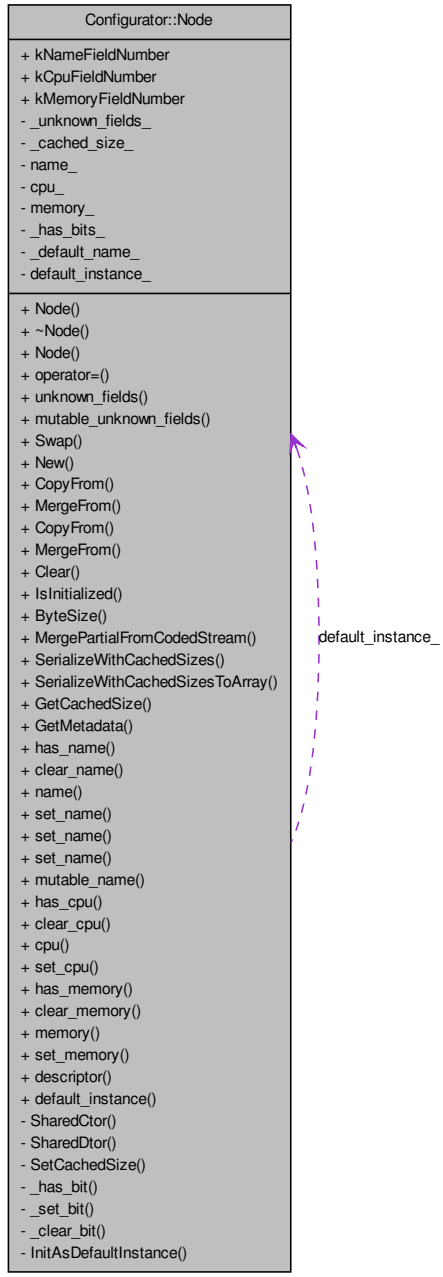
The documentation for this class was generated from the following files:

- Exception.h
- Exception.cpp

7.46 Configurator::Node Class Reference

`#include <confcluster.pb.h>`

Collaboration diagram for Configurator::Node:



7.46.1 Public Member Functions

- Node ()
- virtual ~Node ()
- Node (const Node &from)
- Node & operator= (const Node &from)
- const ::google::protobuf::UnknownFieldSet & unknown_fields () const
- inline::google::protobuf::UnknownFieldSet * mutable_unknown_fields ()
- void Swap (Node *other)
- Node * New () const
- void CopyFrom (const ::google::protobuf::Message &from)
- void MergeFrom (const ::google::protobuf::Message &from)
- void CopyFrom (const Node &from)
- void MergeFrom (const Node &from)
- void Clear ()
- bool IsInitialized () const
- int ByteSize () const
- bool MergePartialFromCodedStream (::google::protobuf::io::CodedInputStream *input)
- void SerializeWithCachedSizes (::google::protobuf::io::CodedOutputStream *output) const
- ::google::protobuf::uint8 * SerializeWithCachedSizesToArray (::google::protobuf::uint8 *output) const
- int GetCachedSize () const
- ::google::protobuf::Metadata GetMetadata () const
- bool has_name () const
- void clear_name ()
- const ::std::string & name () const
- void set_name (const ::std::string &value)
- void set_name (const char *value)
- void set_name (const char *value, size_t size)
- inline::std::string * mutable_name ()
- bool has_cpu () const
- void clear_cpu ()
- double cpu () const

- void set_cpu (double value)
- bool has_memory () const
- void clear_memory ()
- double memory () const
- void set_memory (double value)

7.46.2 Static Public Member Functions

- static const ::google::protobuf::Descriptor * descriptor ()
- static const Node & default_instance ()

7.46.3 Static Public Attributes

- static const int kNameFieldNumber = 1
- static const int kCpuFieldNumber = 2
- static const int kMemoryFieldNumber = 3

7.46.4 Friends

- void protobuf_AddDesc_confcluster_2eproto ()
- void protobuf_AssignDesc_confcluster_2eproto ()
- void protobuf_ShutdownFile_confcluster_2eproto ()

7.46.5 Constructor & Destructor Documentation

Configurator::Node::Node ( )

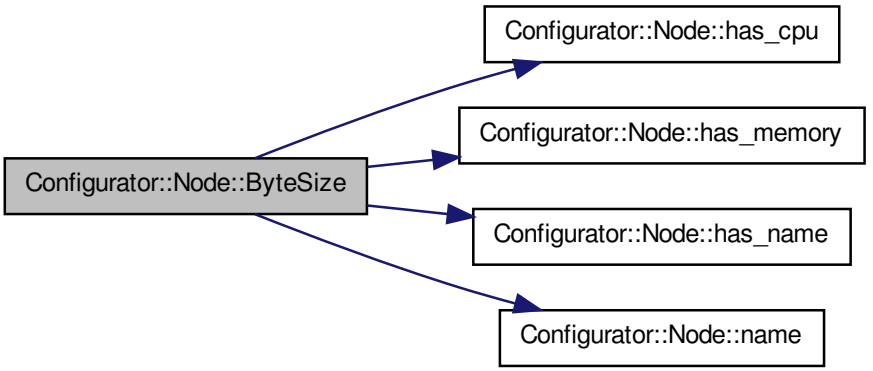
Configurator::Node::~Node ( ) `[virtual]`

Configurator::Node::Node ([const Node &]from)

7.46.6 Member Function Documentation

int Configurator::Node::ByteSize ( ) const

Here is the call graph for this function:



void Configurator::Node::Clear ( )

void Configurator::Node::clear_cpu ( ) [inline]

void Configurator::Node::clear_memory ( ) [inline]

void Configurator::Node::clear_name ( ) [inline]

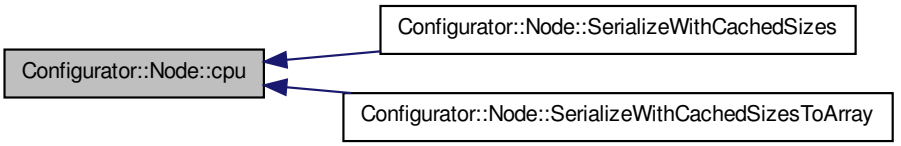
void Configurator::Node::CopyFrom ([const Node &]from)

void Configurator::Node::CopyFrom ([const ::google::protobuf::Message &]from)

Here is the caller graph for this function:

double Configurator::Node::cpu ( ) const [inline]

Here is the caller graph for this function:



const Node & Configurator::Node::default_instance ( ) [static]

const ::google::protobuf::Descriptor * Configurator::Node::descriptor ( ) [static]

int Configurator::Node::GetCachedSize ( ) const [inline]

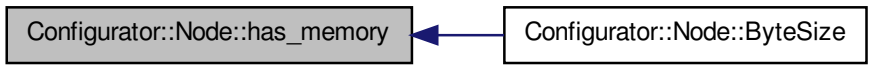
google::protobuf::Metadata Configurator::Node::GetMetadata ( ) const

bool Configurator::Node::has_cpu ( ) const [inline]

Here is the caller graph for this function:

bool Configurator::Node::has_memory ( ) const [inline]

Here is the caller graph for this function:



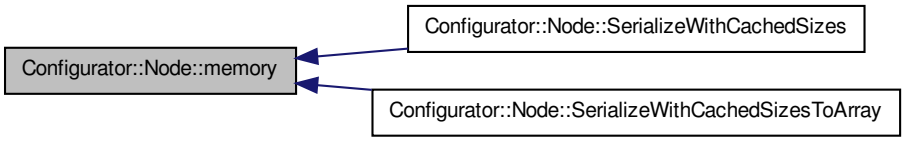
bool Configurator::Node::has_name ( ) const [inline]

Here is the caller graph for this function:

bool Configurator::Node::IsInitialized ( ) const

double Configurator::Node::memory ( ) const [inline]

Here is the caller graph for this function:

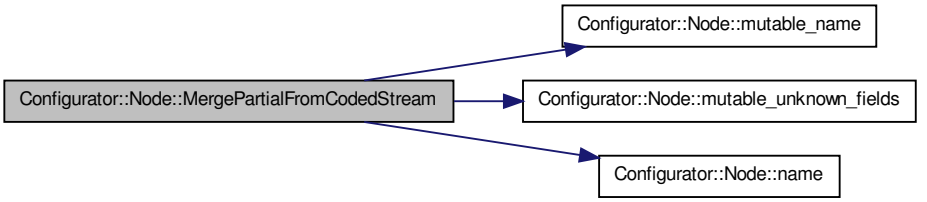


void Configurator::Node::MergeFrom ([const ::google::protobuf::Message &]from)

void Configurator::Node::MergeFrom ([const Node &]from)

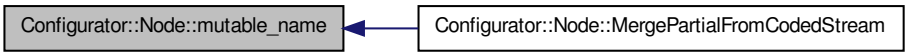
bool Configurator::Node::MergePartialFromCodedStream ([::google::protobuf::io::CodedInputStream ∗]input)

Here is the call graph for this function:



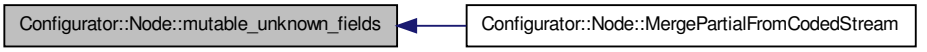
std::string ∗ Configurator::Node::mutable_name ( ) `[inline]`

Here is the caller graph for this function:



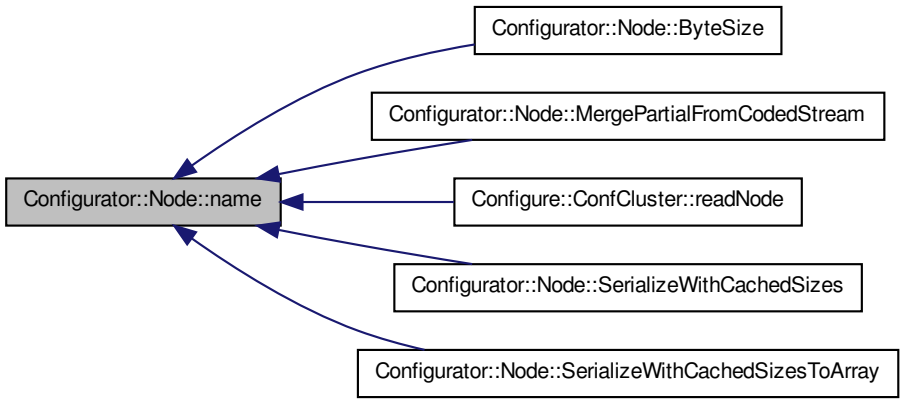
inline ::google::protobuf::UnknownFieldSet∗ Configurator::Node::mutable_-unknown_fields ( ) `[inline]`

Here is the caller graph for this function:



const ::std::string & Configurator::Node::name ( ) const `[inline]`

Here is the caller graph for this function:



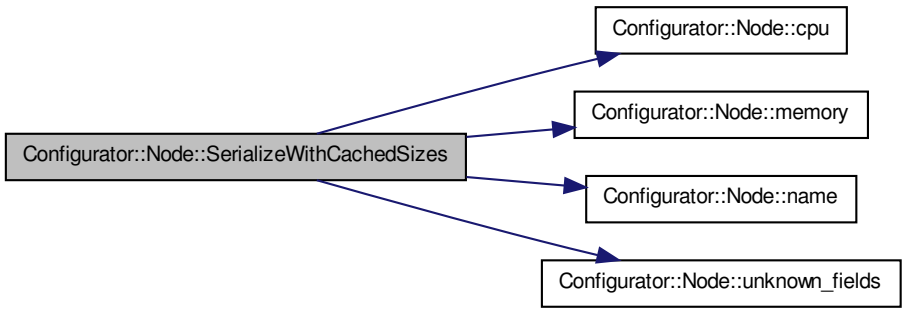
Node * Configurator::Node::New ( ) const

Node& Configurator::Node::operator= ([const Node &]from) `[inline]`

Here is the call graph for this function:

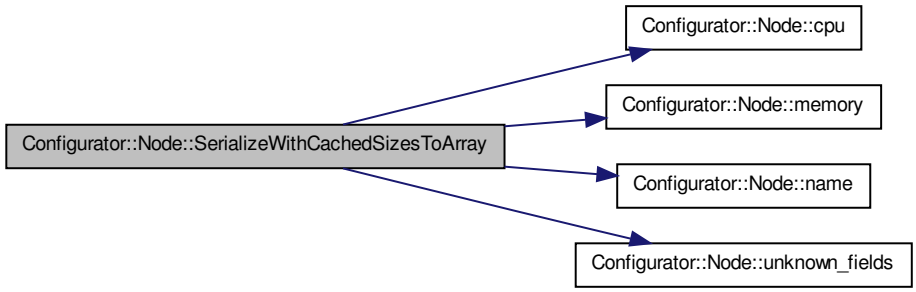
void Configurator::Node::SerializeWithCachedSizes ([::google::protobuf::io::CodedOutputStream *]output) const

Here is the call graph for this function:



google::protobuf::uint8 * Configurator::Node::SerializeWithCachedSizesToArray ([::google::protobuf::uint8 *]output) const

Here is the call graph for this function:



void Configurator::Node::set_cpu ([double]value) [inline]

void Configurator::Node::set_memory ([double]value) [inline]

void Configurator::Node::set_name ([const char *]value, size_t *size*) [inline]

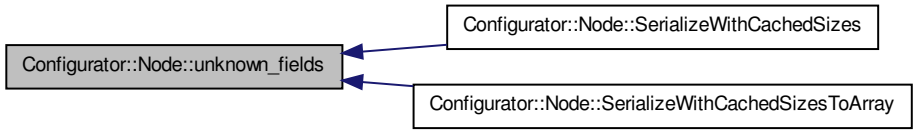
void Configurator::Node::set_name ([const char *]value) [inline]

void Configurator::Node::set_name ([const ::std::string &]value) [inline]

void Configurator::Node::Swap ([Node *]other)

const ::google::protobuf::UnknownFieldSet& Configurator::Node::unknown_fields ( ) const [inline]

Here is the caller graph for this function:



7.46.7 Friends And Related Function Documentation

void protobuf_AddDesc_confcluster_2eproto ( ) [friend]

void protobuf_AssignDesc_confcluster_2eproto ( ) [friend]

void protobuf_ShutdownFile_confcluster_2eproto ( ) [friend]

7.46.8 Member Data Documentation

const int Configurator::Node::kCpuFieldNumber = 2 `[static]`

const int Configurator::Node::kMemoryFieldNumber = 3 `[static]`

const int Configurator::Node::kNameFieldNumber = 1 `[static]`

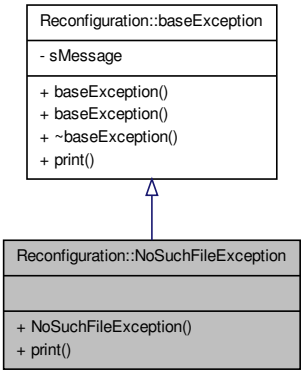
The documentation for this class was generated from the following files:

- confcluster.pb.h
- confcluster.pb.cc

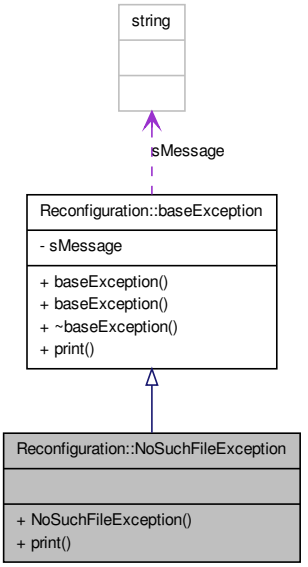
7.47 Reconfiguration::NoSuchFileException Class Reference

`#include <Exception.h>`

Inheritance diagram for Reconfiguration::NoSuchFileException:



Collaboration diagram for Reconfiguration::NoSuchFileException:



7.47.1 Public Member Functions

- NoSuchFileException (std::string)

  Constructor of the NoSuchFileException class.

- void print ()

  Prints the NoSuchFileException exception.

7.47.2 Detailed Description

Exception thrown if an nonexistent file is requested.

7.47.3 Constructor & Destructor Documentation

Reconfiguration::NoSuchFileException::NoSuchFileException ([std::string]sFile)

Constructor of the NoSuchFileException class.

Parameters

in	*sFile*	Requested filename.

Returns

∗this

Creates an object NoSuchFileException calling the baseException constructor and sets the message of the exception.

7.47.4 Member Function Documentation

void Reconfiguration::NoSuchFileException::print ( )

Prints the NoSuchFileException exception.

Returns

void

Prints the exception by printing its message.

Reimplemented from Reconfiguration::baseException.

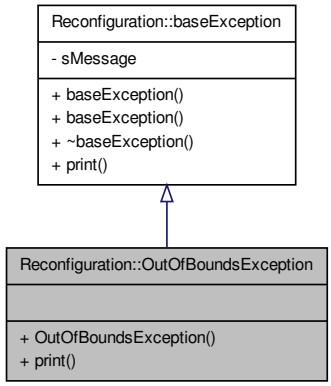
The documentation for this class was generated from the following files:

- Exception.h
- Exception.cpp

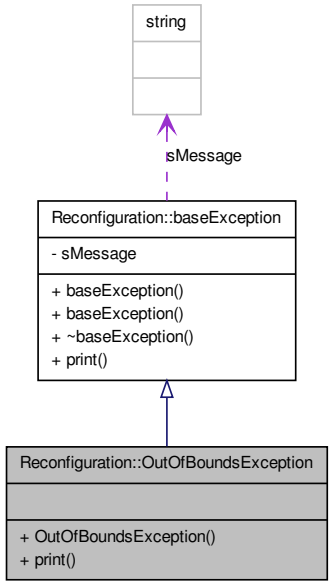
7.48 Reconfiguration::OutOfBoundsException Class Reference

`#include <Exception.h>`

Inheritance diagram for Reconfiguration::OutOfBoundsException:



Collaboration diagram for Reconfiguration::OutOfBoundsException:



7.48.1 Public Member Functions

- OutOfBoundsException (int)

  Constructor of the OutOfBoundsException class.

- void print ()

  Prints the OutOfBoundsException exception.

7.48.2 Detailed Description

Exception thrown if the limit of an structure is exceeded.

7.48.3 Constructor & Destructor Documentation

Reconfiguration::OutOfBoundsException::OutOfBoundsException ([int]iOffset)

Constructor of the OutOfBoundsException class.

Parameters

in	*iOffset*	Offset reached and exceeded.

Returns

∗this

Creates an object OutOfBoundsException calling the baseException constructor and sets the message of the exception.

Here is the call graph for this function:

7.48.4 Member Function Documentation

void Reconfiguration::OutOfBoundsException::print ( )

Prints the OutOfBoundsException exception.

Returns

void

Prints the exception by printing its message.

Reimplemented from Reconfiguration::baseException.

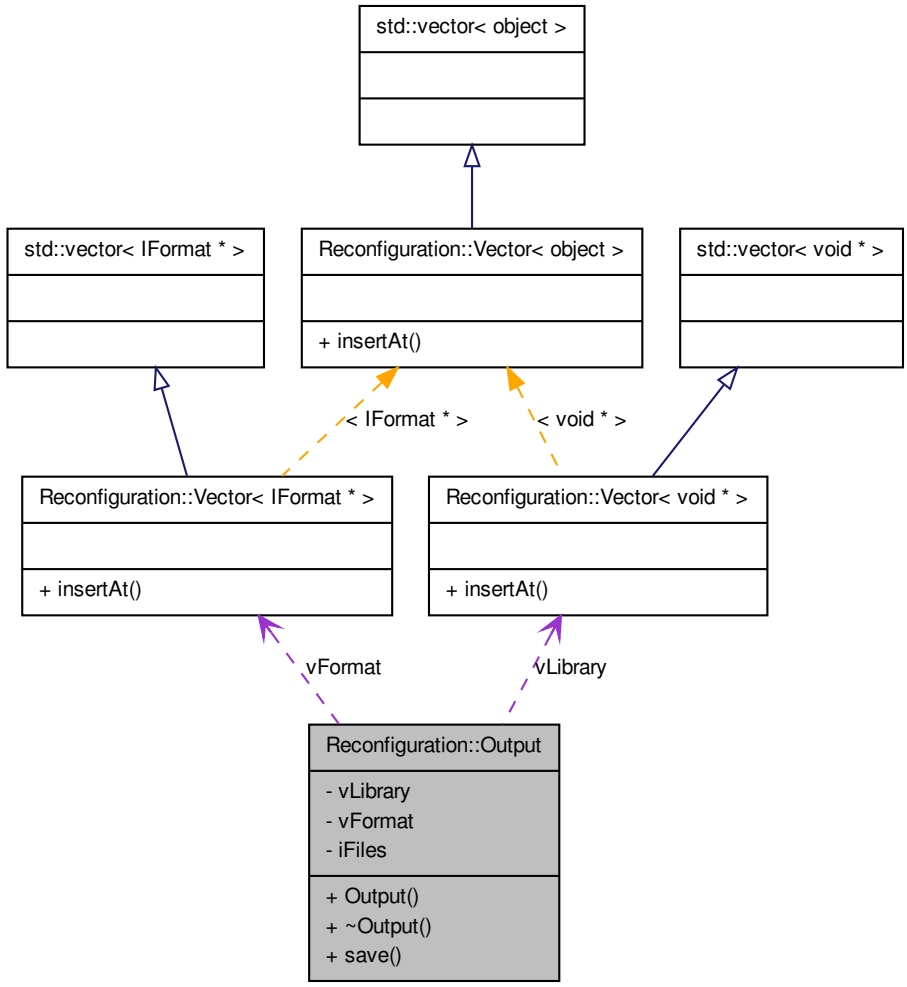
The documentation for this class was generated from the following files:

- Exception.h
- Exception.cpp

7.49 Reconfiguration::Output Class Reference

`#include <Output.h>`

Collaboration diagram for Reconfiguration::Output:



7.49.1 Public Member Functions

- Output (int, std::string *, std::string, int, int *, bool)

  Constructor of the Output class.

- virtual ~Output ()

  virtual destructor of the class Output.

- void save (std::stringstream ∗, bool)

  Saves the step results into a file.

7.49.2 Detailed Description

Manages the output plugins (dynamic libraries) called by the user and its handlers.

7.49.3 Constructor & Destructor Documentation

Reconfiguration::Output::Output ([int]iFormats, std::string ∗ *sFormatNames,* std::string *sName,* int *iDimension,* int ∗ *iDimensions,* bool *savePartition*)

Constructor of the Output class.

Parameters

in	*iFormats*	Number of plugins specified by the user.
in	*sFormatNames*	Names of the plugins (without extension and path).
in	*sName*	Base name of the results files.
in	*iDimension*	Number of dimensions of the problem.
in	*iDimensions*	Length per dimension.
in	*savePartition*	If true, the platform generates output files for the partition shapes. Otherwise, the partitions are not saved as PNG files.

Returns

∗this

Creates an object Output. The output plugins must "live" at lib/output/ and its name must begin with lib (ie: plugin A, dynamic library lib/output/libA.so).

Exceptions

In	case of error, the InternalException exception is thrown.

Note

If the partitions are saved as PNG, the iFiles field is not incremented.

Here is the call graph for this function:

Reconfiguration::Output::~Output ( ) [virtual]

virtual destructor of the class Output.

Returns

void

Destroys the Output object.

7.49.4 Member Function Documentation

void Reconfiguration::Output::save ([std::stringstream *]ss, bool *partitions*)

Saves the step results into a file.

Parameters

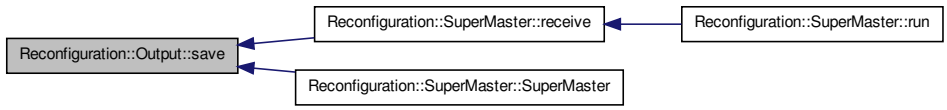
in	*ss*	Results of the execution of the step, formated as the user wants.
in	*partitions*	Indicates whether the partition shapes must be saved into a PNG file.

Returns

void

Save the results of a simulation step into a file, that can be either an image or a plain text file, following the format chosen by the user.

Here is the caller graph for this function:



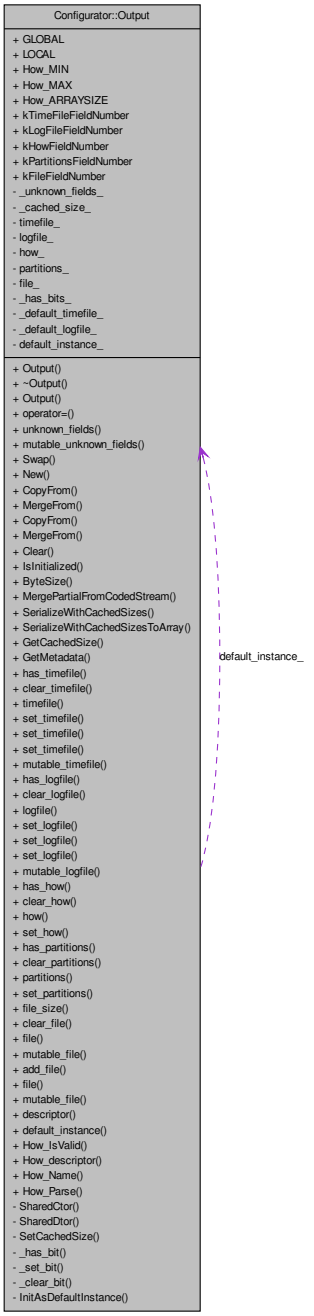
The documentation for this class was generated from the following files:

- Output.h
- Output.cpp

7.50 Configurator::Output Class Reference

`#include <confproblem.pb.h>`

Collaboration diagram for Configurator::Output:



7.50.1 Public Types

- typedef Output_How How

7.50.2 Public Member Functions

- Output ()
- virtual ~Output ()
- Output (const Output &from)
- Output & operator= (const Output &from)
- const ::google::protobuf::UnknownFieldSet & unknown_fields () const
- inline::google::protobuf::UnknownFieldSet * mutable_unknown_fields ()
- void Swap (Output *other)
- Output * New () const
- void CopyFrom (const ::google::protobuf::Message &from)
- void MergeFrom (const ::google::protobuf::Message &from)
- void CopyFrom (const Output &from)
- void MergeFrom (const Output &from)
- void Clear ()
- bool IsInitialized () const
- int ByteSize () const
- bool MergePartialFromCodedStream (::google::protobuf::io::CodedInputStream *input)
- void SerializeWithCachedSizes (::google::protobuf::io::CodedOutputStream *output) const
- ::google::protobuf::uint8 * SerializeWithCachedSizesToArray (::google::protobuf::uint8 *output) const
- int GetCachedSize () const
- ::google::protobuf::Metadata GetMetadata () const
- bool has_timefile () const
- void clear_timefile ()
- const ::std::string & timefile () const
- void set_timefile (const ::std::string &value)
- void set_timefile (const char *value)
- void set_timefile (const char *value, size_t size)

- inline::std::string * mutable_timefile ()
- bool has_logfile () const
- void clear_logfile ()
- const ::std::string & logfile () const
- void set_logfile (const ::std::string &value)
- void set_logfile (const char *value)
- void set_logfile (const char *value, size_t size)
- inline::std::string * mutable_logfile ()
- bool has_how () const
- void clear_how ()
- inline::Configurator::Output_How how () const
- void set_how (::Configurator::Output_How value)
- bool has_partitions () const
- void clear_partitions ()
- bool partitions () const
- void set_partitions (bool value)
- int file_size () const
- void clear_file ()
- const ::Configurator::OutputFormat & file (int index) const
- inline::Configurator::OutputFormat * mutable_file (int index)
- inline::Configurator::OutputFormat * add_file ()
- const ::google::protobuf::RepeatedPtrField< ::Configurator::OutputFormat > & file () const
- inline::google::protobuf::RepeatedPtrField< ::Configurator::OutputFormat > * mutable_file ()

7.50.3 Static Public Member Functions

- static const ::google::protobuf::Descriptor * descriptor ()
- static const Output & default_instance ()
- static bool How_IsValid (int value)
- static const ::google::protobuf::EnumDescriptor * How_descriptor ()
- static const ::std::string & How_Name (How value)

- static bool How_Parse (const ::std::string &name, How *value)

7.50.4 Static Public Attributes

- static const How GLOBAL = Output_How_GLOBAL
- static const How LOCAL = Output_How_LOCAL
- static const How How_MIN
- static const How How_MAX
- static const int How_ARRAYSIZE
- static const int kTimeFileFieldNumber = 1
- static const int kLogFileFieldNumber = 2
- static const int kHowFieldNumber = 3
- static const int kPartitionsFieldNumber = 4
- static const int kFileFieldNumber = 5

7.50.5 Friends

- void protobuf_AddDesc_confproblem_2eproto ()
- void protobuf_AssignDesc_confproblem_2eproto ()
- void protobuf_ShutdownFile_confproblem_2eproto ()

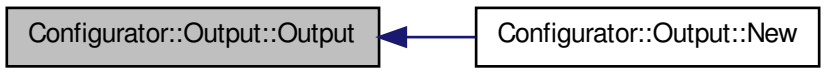
7.50.6 Member Typedef Documentation

typedef Output_How Configurator::Output::How

7.50.7 Constructor & Destructor Documentation

Configurator::Output::Output ( )

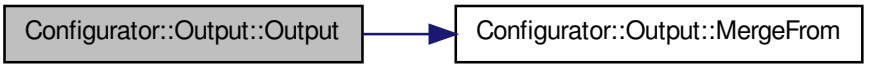
Here is the caller graph for this function:



Configurator::Output::~Output ( ) [virtual]

Configurator::Output::Output ([const Output &]from)

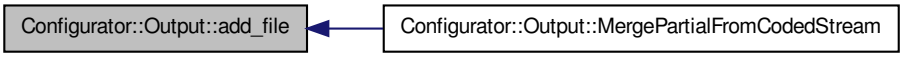
Here is the call graph for this function:



7.50.8 Member Function Documentation

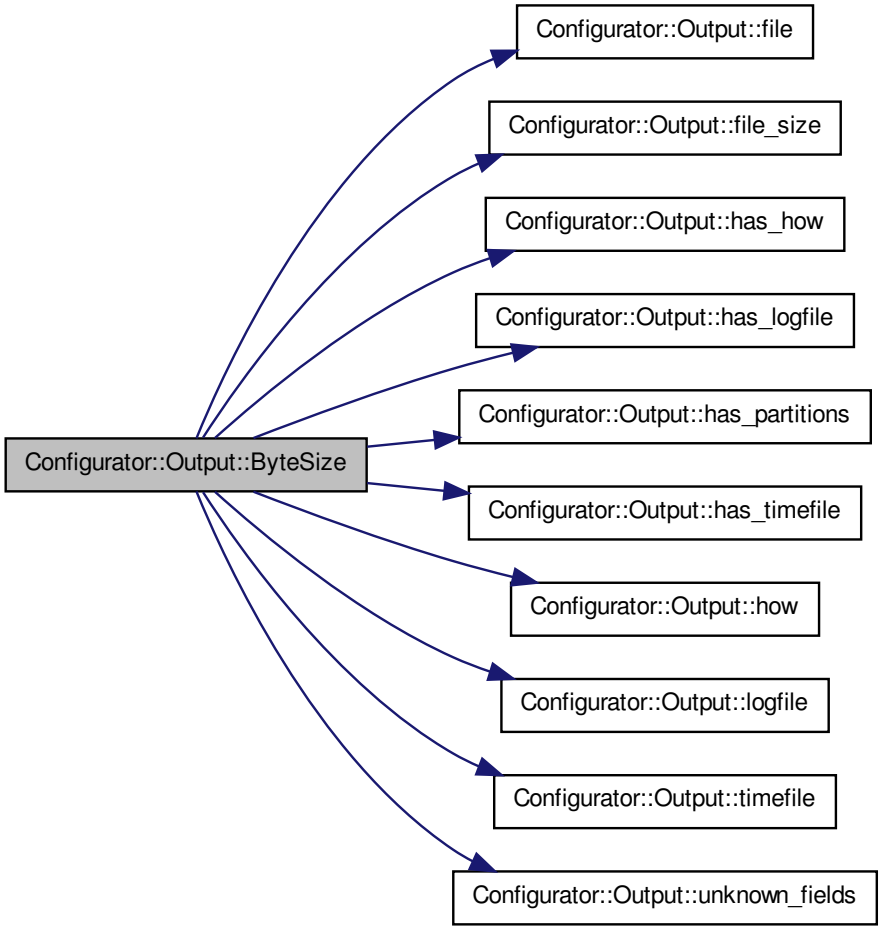
Configurator::OutputFormat * Configurator::Output::add_file ( ) [inline]

Here is the caller graph for this function:



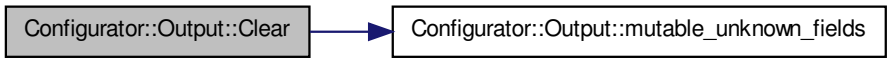
int Configurator::Output::ByteSize ( ) const

Here is the call graph for this function:

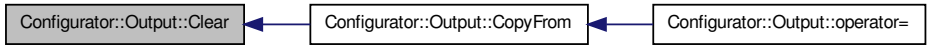


void Configurator::Output::Clear ( )

Here is the call graph for this function:



Here is the caller graph for this function:



void Configurator::Output::clear_file ( ) [inline]

void Configurator::Output::clear_how ( ) [inline]

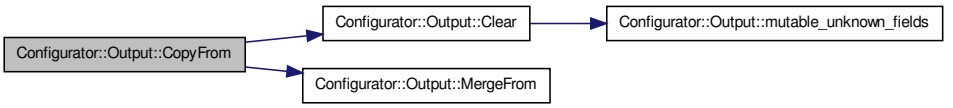
void Configurator::Output::clear_logfile ( ) [inline]

void Configurator::Output::clear_partitions ( ) [inline]

void Configurator::Output::clear_timefile ( ) [inline]

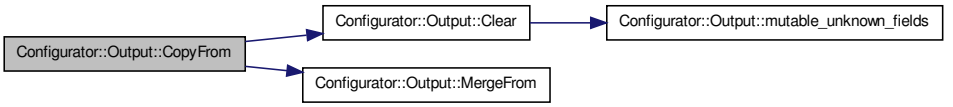
void Configurator::Output::CopyFrom ([const Output &]from)

Here is the call graph for this function:



void Configurator::Output::CopyFrom ([const ::google::protobuf::Message &]from)

Here is the call graph for this function:



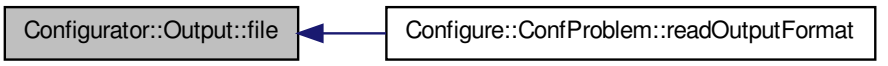
Here is the caller graph for this function:

const Output & Configurator::Output::default_instance ( ) [static]

const ::google::protobuf::Descriptor * Configurator::Output::descriptor ( ) [static]

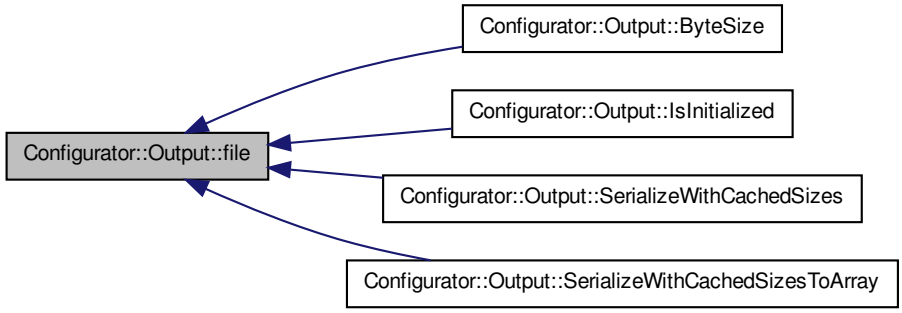
const ::Configurator::OutputFormat & Configurator::Output::file ([int]index) const [inline]

Here is the caller graph for this function:



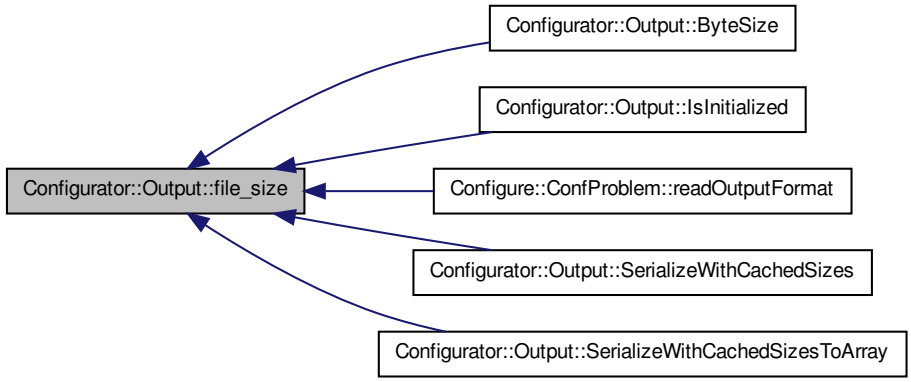
const ::google::protobuf::RepeatedPtrField<::Configurator::OutputFormat > & Configurator::Output::file ( ) const [inline]

Here is the caller graph for this function:



int Configurator::Output::file_size ( ) const [inline]

Here is the caller graph for this function:



int Configurator::Output::GetCachedSize ( ) const [inline]

google::protobuf::Metadata Configurator::Output::GetMetadata ( ) const

bool Configurator::Output::has_how ( ) const [inline]

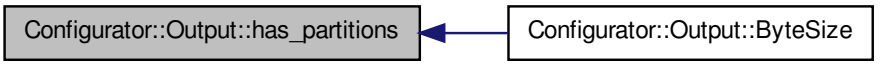
Here is the caller graph for this function:

bool Configurator::Output::has_logfile ( ) const [inline]

Here is the caller graph for this function:

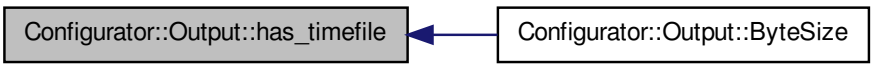
bool Configurator::Output::has_partitions ( ) const [inline]

Here is the caller graph for this function:



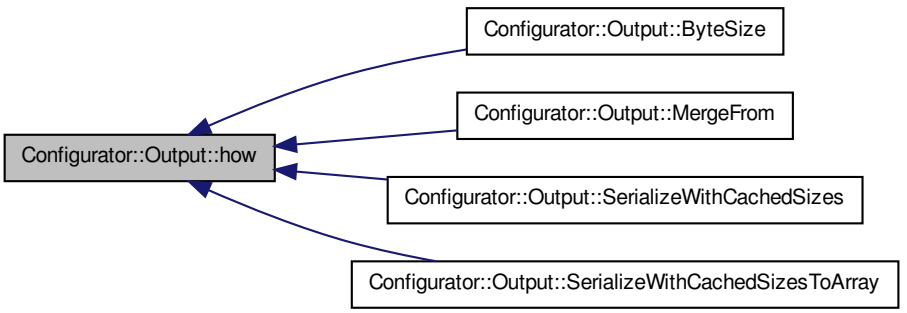
bool Configurator::Output::has_timefile ( ) const [inline]

Here is the caller graph for this function:



Configurator::Output_How Configurator::Output::how ( ) const [inline]

Here is the caller graph for this function:



static const ::google::protobuf::EnumDescriptor* Configurator::Output::How_-descriptor ( ) [inline, static]

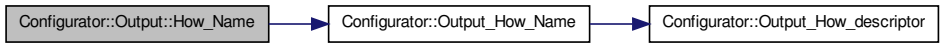
Here is the call graph for this function:

static bool Configurator::Output::How_IsValid ([int]value) `[inline, static]`

Here is the call graph for this function:

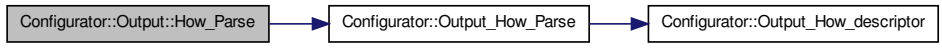
static const ::std::string& Configurator::Output::How_Name ([How]value) `[inline, static]`

Here is the call graph for this function:



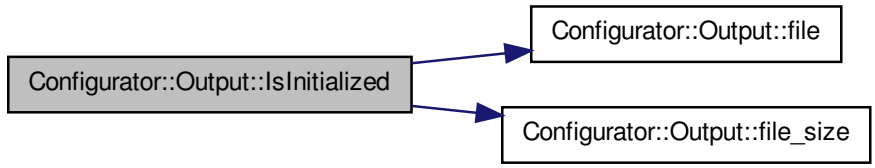
static bool Configurator::Output::How_Parse ([const ::std::string &]name, How * *value*) `[inline, static]`

Here is the call graph for this function:



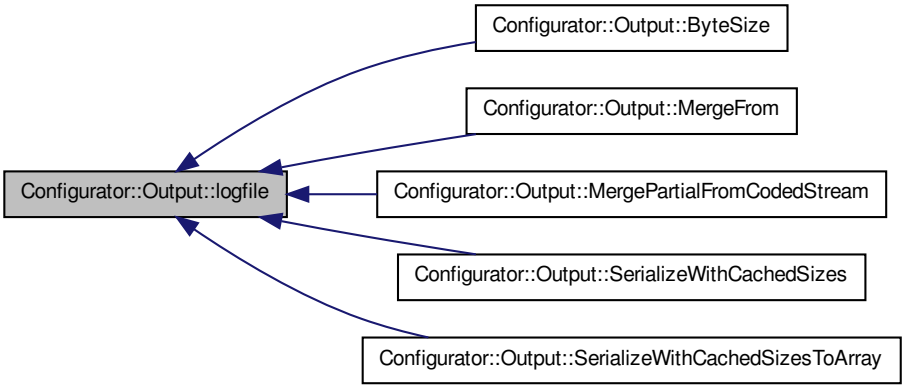
bool Configurator::Output::IsInitialized ( ) const

Here is the call graph for this function:



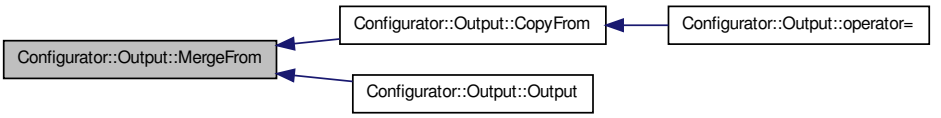
const ::std::string & Configurator::Output::logfile ( ) const `[inline]`

Here is the caller graph for this function:



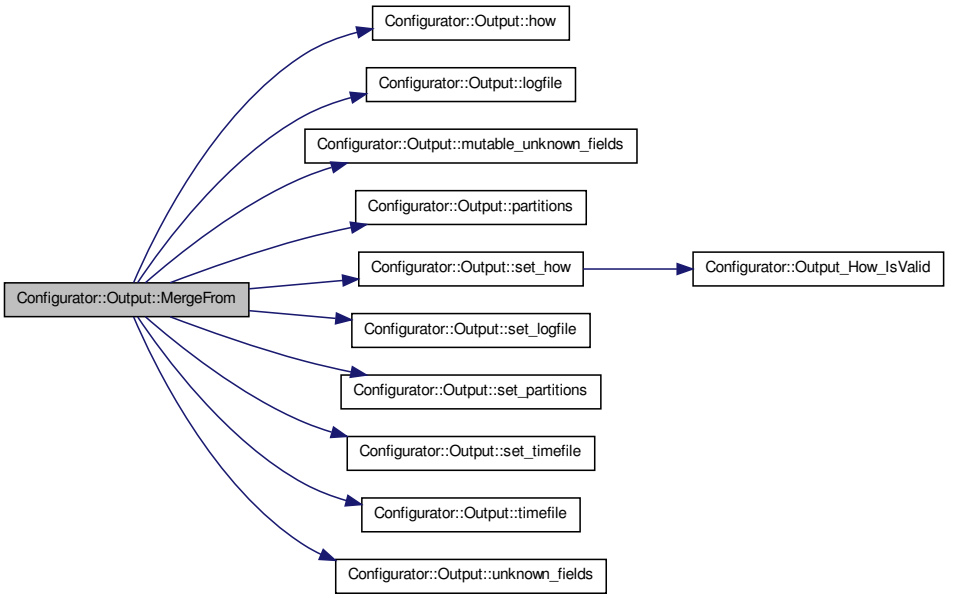
void Configurator::Output::MergeFrom ([const ::google::protobuf::Message &]from)

Here is the caller graph for this function:



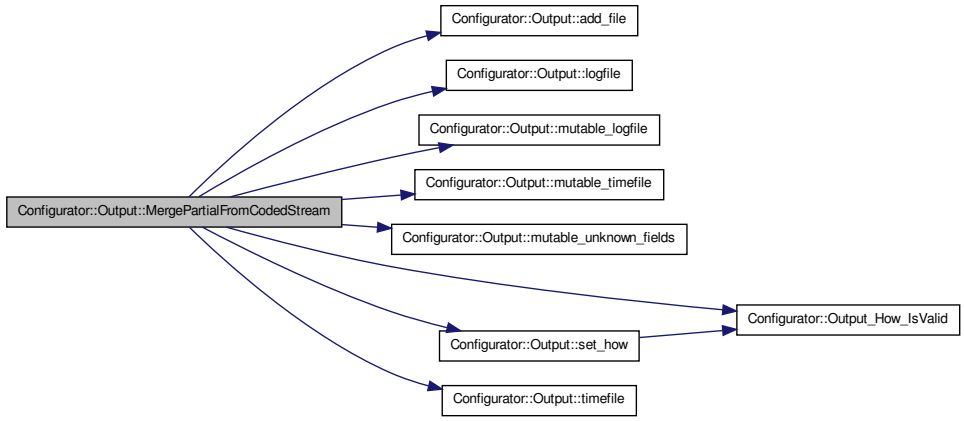
void Configurator::Output::MergeFrom ([const Output &]from)

Here is the call graph for this function:



bool Configurator::Output::MergePartialFromCodedStream ([::google::protobuf::io::CodedInputStream ∗]input)

Here is the call graph for this function:

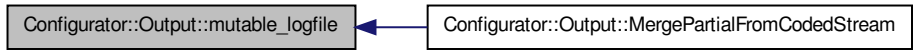


Configurator::OutputFormat ∗ Configurator::Output::mutable_file ([int]index) [inline]

google::protobuf::RepeatedPtrField<::Configurator::OutputFormat > ∗ Configurator::Output::mutable_file ( ) [inline]

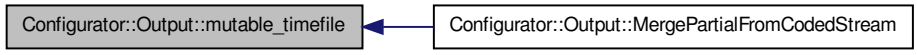
std::string ∗ Configurator::Output::mutable_logfile ( ) [inline]

Here is the caller graph for this function:



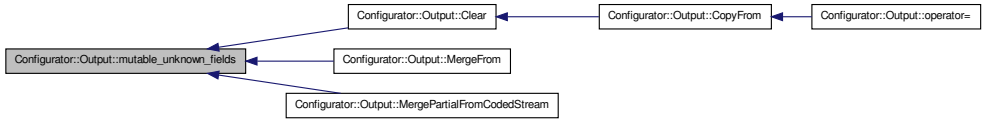
std::string ∗ Configurator::Output::mutable_timefile ( ) [inline]

Here is the caller graph for this function:



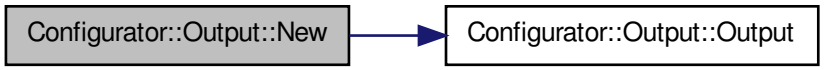
inline ::google::protobuf::UnknownFieldSet∗ Configurator::Output::mutable_-unknown_fields ( ) [inline]

Here is the caller graph for this function:



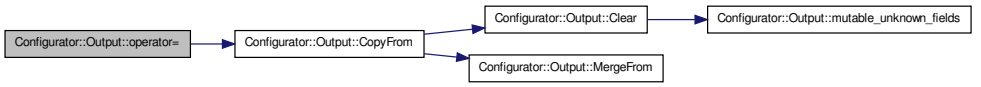
Output * Configurator::Output::New ( ) const

Here is the call graph for this function:



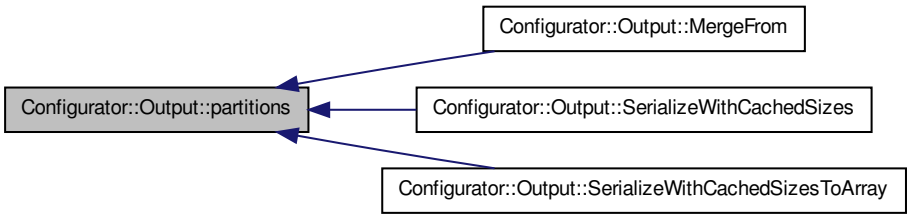
Output& Configurator::Output::operator= ([const Output &]from) [inline]

Here is the call graph for this function:



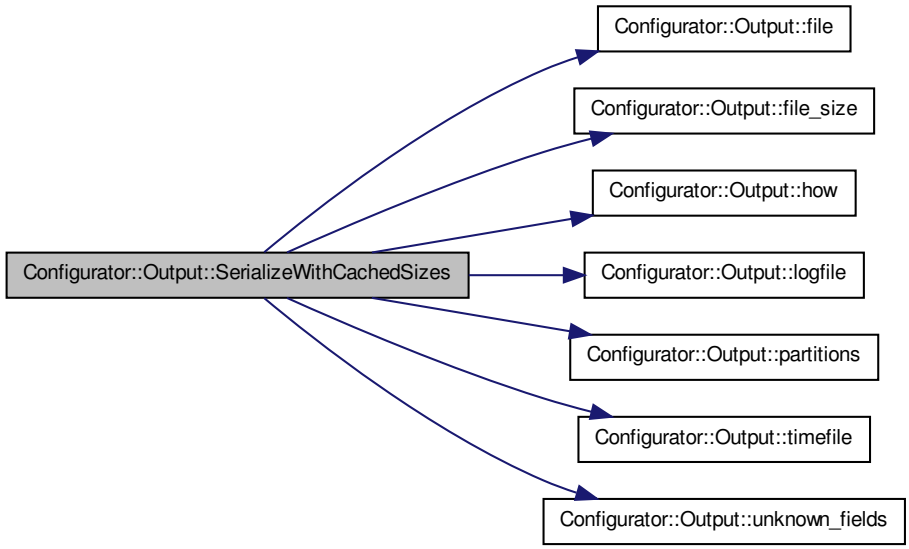
bool Configurator::Output::partitions ( ) const [inline]

Here is the caller graph for this function:



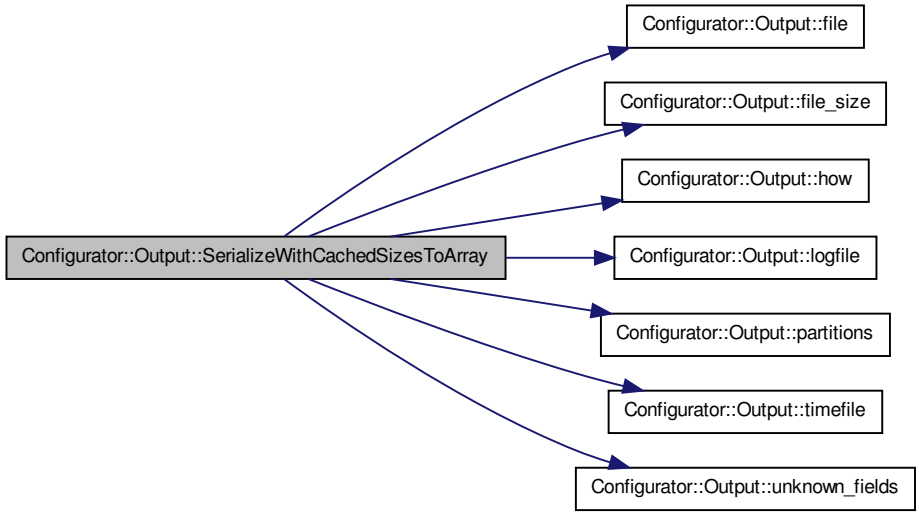
void Configurator::Output::SerializeWithCachedSizes ([::google::protobuf::io::CodedOutputStream *]output) const

Here is the call graph for this function:



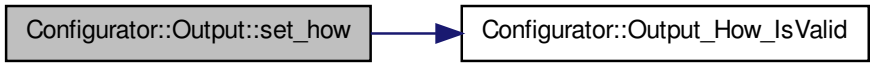
google::protobuf::uint8 * Configurator::Output::SerializeWithCachedSizesToArray ([::google::protobuf::uint8 *]output) const

Here is the call graph for this function:

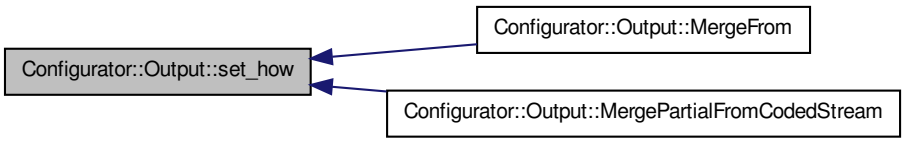


void Configurator::Output::set_how ([::Configurator::Output_How]value) `[inline]`

Here is the call graph for this function:



Here is the caller graph for this function:



void Configurator::Output::set_logfile ([const char *]value) [inline]

void Configurator::Output::set_logfile ([const char *]value, size_t *size*) [inline]

void Configurator::Output::set_logfile ([const ::std::string &]value) [inline]

Here is the caller graph for this function:

void Configurator::Output::set_partitions ([bool]value) [inline]

Here is the caller graph for this function:

void Configurator::Output::set_timefile ([const char *]value, size_t *size*) [inline]

void Configurator::Output::set_timefile ([const char *]value) [inline]

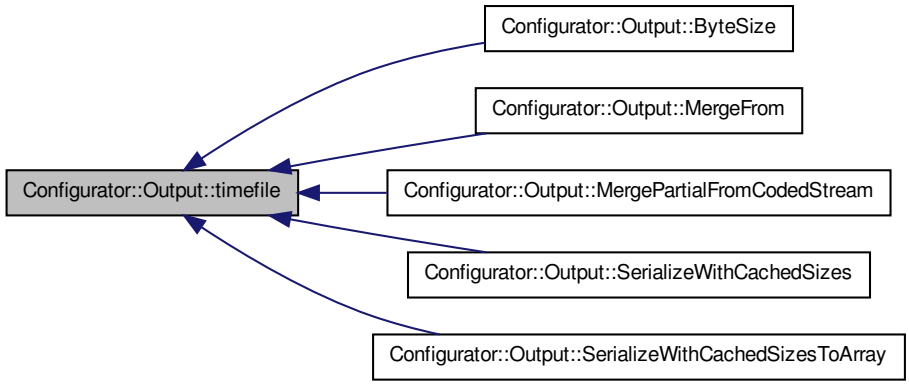
void Configurator::Output::set_timefile ([const ::std::string &]value) [inline]

Here is the caller graph for this function:

void Configurator::Output::Swap ([Output *]other)

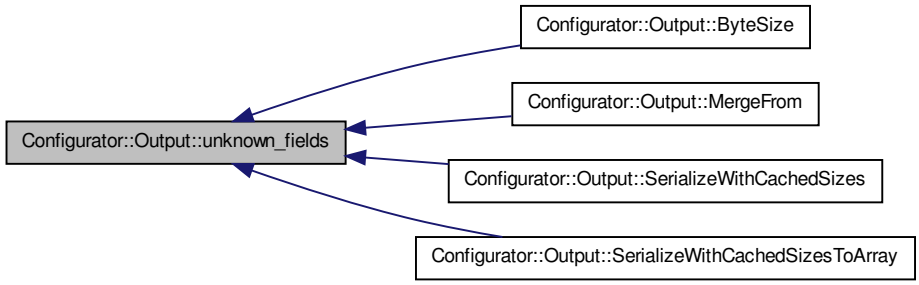
const ::std::string & Configurator::Output::timefile ( ) const [inline]

Here is the caller graph for this function:



const ::google::protobuf::UnknownFieldSet& Configurator::Output::unknown_fields ( ) const [inline]

Here is the caller graph for this function:



7.50.9 Friends And Related Function Documentation

void protobuf_AddDesc_confproblem_2eproto ( ) [friend]

void protobuf_AssignDesc_confproblem_2eproto ( ) [friend]

void protobuf_ShutdownFile_confproblem_2eproto ( ) [friend]

7.50.10 Member Data Documentation

const Output_How Configurator::Output::GLOBAL = Output_How_GLOBAL [static]

const int Configurator::Output::How_ARRAYSIZE [static]

Initial value:

```
Output_How_How_ARRAYSIZE
```

const Output_How Configurator::Output::How_MAX [static]

Initial value:

```
Output_How_How_MAX
```

const Output_How Configurator::Output::How_MIN [static]

Initial value:

```
Output_How_How_MIN
```

const int Configurator::Output::kFileFieldNumber = 5 `[static]`

const int Configurator::Output::kHowFieldNumber = 3 `[static]`

const int Configurator::Output::kLogFileFieldNumber = 2 `[static]`

const int Configurator::Output::kPartitionsFieldNumber = 4 `[static]`

const int Configurator::Output::kTimeFileFieldNumber = 1 `[static]`

const Output_How Configurator::Output::LOCAL = Output_How_LOCAL `[static]`

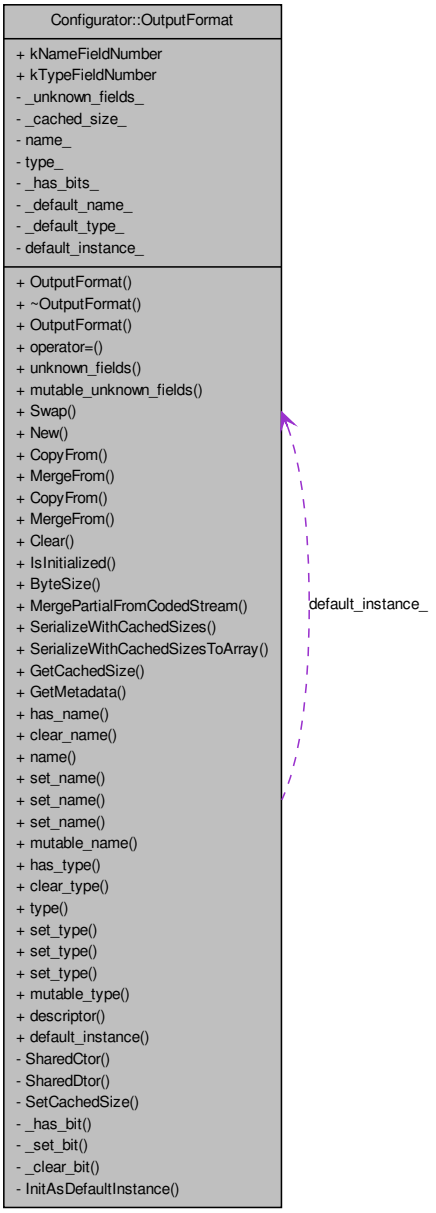
The documentation for this class was generated from the following files:

- confproblem.pb.h
- confproblem.pb.cc

7.51 Configurator::OutputFormat Class Reference

`#include <confproblem.pb.h>`

Collaboration diagram for Configurator::OutputFormat:



7.51.1 Public Member Functions

- OutputFormat ()
- virtual ~OutputFormat ()
- OutputFormat (const OutputFormat &from)
- OutputFormat & operator= (const OutputFormat &from)
- const ::google::protobuf::UnknownFieldSet & unknown_fields () const
- inline::google::protobuf::UnknownFieldSet * mutable_unknown_fields ()
- void Swap (OutputFormat *other)
- OutputFormat * New () const
- void CopyFrom (const ::google::protobuf::Message &from)
- void MergeFrom (const ::google::protobuf::Message &from)
- void CopyFrom (const OutputFormat &from)
- void MergeFrom (const OutputFormat &from)
- void Clear ()
- bool IsInitialized () const
- int ByteSize () const
- bool MergePartialFromCodedStream (::google::protobuf::io::CodedInputStream *input)
- void SerializeWithCachedSizes (::google::protobuf::io::CodedOutputStream *output) const
- ::google::protobuf::uint8 * SerializeWithCachedSizesToArray (::google::protobuf::uint8 *output) const
- int GetCachedSize () const
- ::google::protobuf::Metadata GetMetadata () const
- bool has_name () const
- void clear_name ()
- const ::std::string & name () const
- void set_name (const ::std::string &value)
- void set_name (const char *value)
- void set_name (const char *value, size_t size)
- inline::std::string * mutable_name ()
- bool has_type () const
- void clear_type ()
- const ::std::string & type () const

- void set_type (const ::std::string &value)
- void set_type (const char *value)
- void set_type (const char *value, size_t size)
- inline::std::string * mutable_type ()

7.51.2 Static Public Member Functions

- static const ::google::protobuf::Descriptor * descriptor ()
- static const OutputFormat & default_instance ()

7.51.3 Static Public Attributes

- static const int kNameFieldNumber = 1
- static const int kTypeFieldNumber = 2

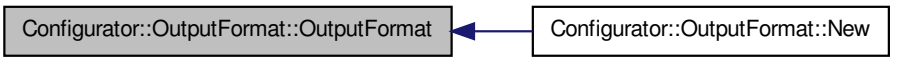
7.51.4 Friends

- void protobuf_AddDesc_confproblem_2eproto ()
- void protobuf_AssignDesc_confproblem_2eproto ()
- void protobuf_ShutdownFile_confproblem_2eproto ()

7.51.5 Constructor & Destructor Documentation

Configurator::OutputFormat::OutputFormat ( )

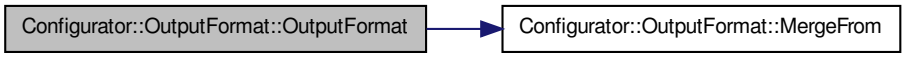
Here is the caller graph for this function:



Configurator::OutputFormat::~OutputFormat ( ) [virtual]

Configurator::OutputFormat::OutputFormat ([const OutputFormat &]from)

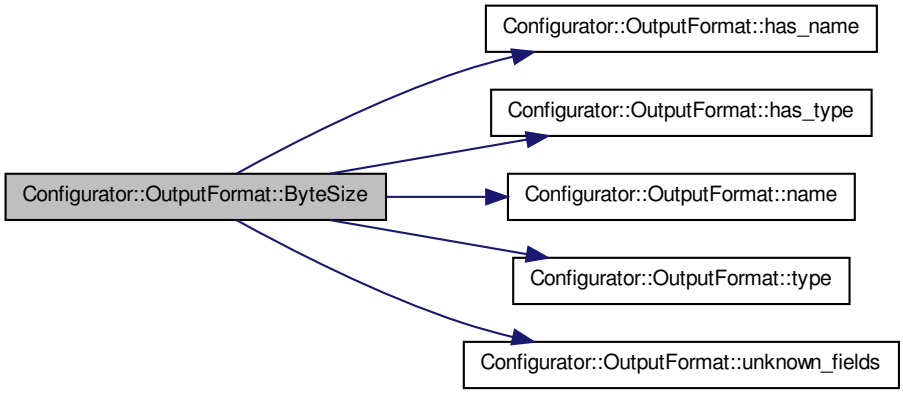
Here is the call graph for this function:



7.51.6 Member Function Documentation

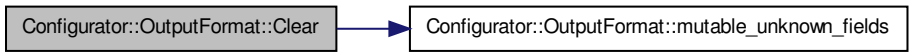
int Configurator::OutputFormat::ByteSize ( ) const

Here is the call graph for this function:

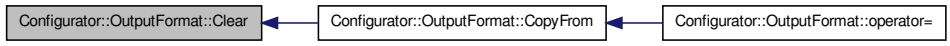


void Configurator::OutputFormat::Clear ( )

Here is the call graph for this function:



Here is the caller graph for this function:

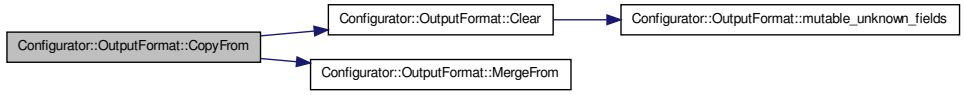


void Configurator::OutputFormat::clear_name ( ) [inline]

void Configurator::OutputFormat::clear_type ( ) [inline]

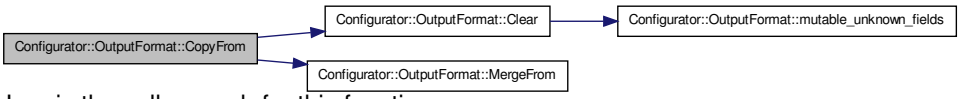
void Configurator::OutputFormat::CopyFrom ([const OutputFormat &]from)

Here is the call graph for this function:



void Configurator::OutputFormat::CopyFrom ([const ::google::protobuf::Message &]from)

Here is the call graph for this function:



Here is the caller graph for this function:

const OutputFormat & Configurator::OutputFormat::default_instance ( ) [static]

const ::google::protobuf::Descriptor * Configurator::OutputFormat::descriptor ( ) [static]

int Configurator::OutputFormat::GetCachedSize ( ) const [inline]

google::protobuf::Metadata Configurator::OutputFormat::GetMetadata ( ) const

bool Configurator::OutputFormat::has_name ( ) const [inline]

Here is the caller graph for this function:

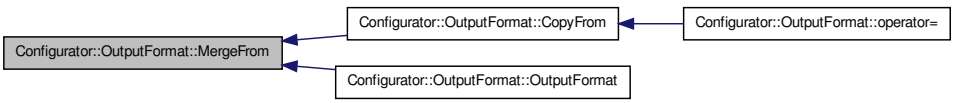
bool Configurator::OutputFormat::has_type ( ) const [inline]

Here is the caller graph for this function:

bool Configurator::OutputFormat::IsInitialized ( ) const

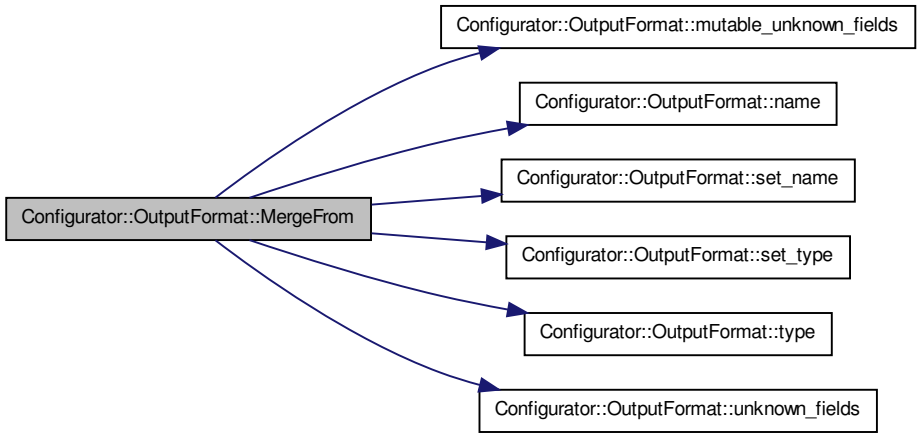
void Configurator::OutputFormat::MergeFrom ([const ::google::protobuf::Message &]from)

Here is the caller graph for this function:



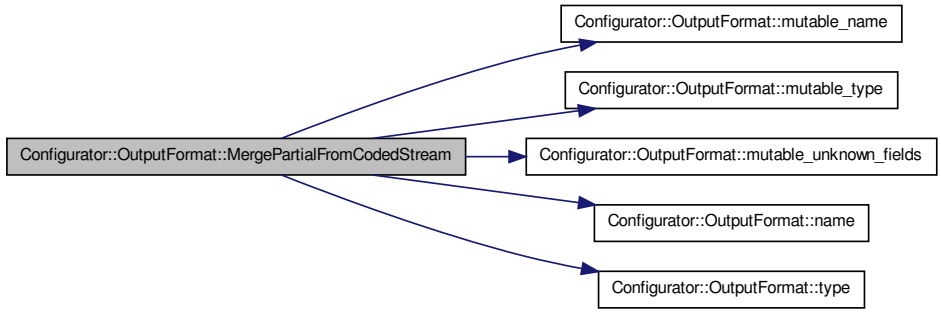
void Configurator::OutputFormat::MergeFrom ([const OutputFormat &]from)

Here is the call graph for this function:



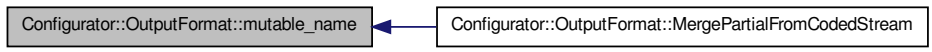
bool Configurator::OutputFormat::MergePartialFromCodedStream ([::google::protobuf::io::CodedInputStream *]input)

Here is the call graph for this function:



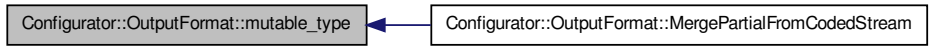
std::string * Configurator::OutputFormat::mutable_name ( ) [inline]

Here is the caller graph for this function:



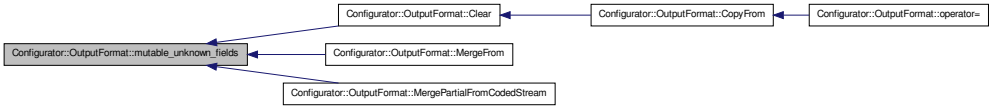
std::string * Configurator::OutputFormat::mutable_type ( ) [inline]

Here is the caller graph for this function:



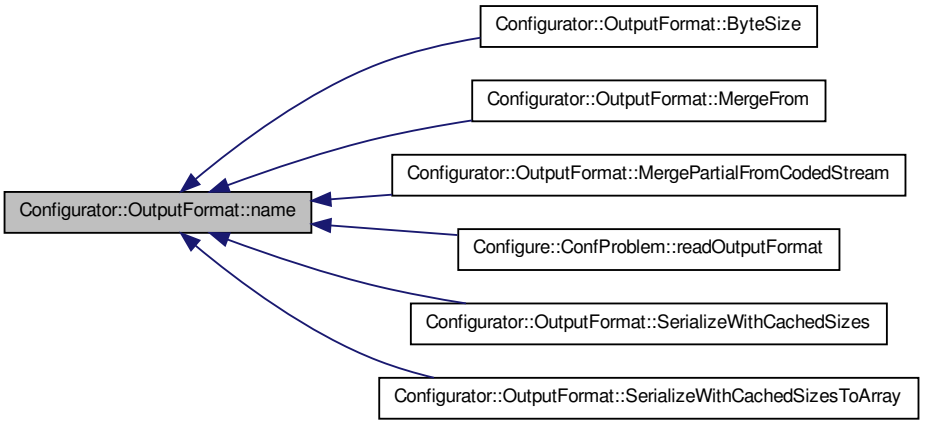
inline ::google::protobuf::UnknownFieldSet* Configurator::OutputFormat::mutable_unknown_fields ( ) [inline]

Here is the caller graph for this function:



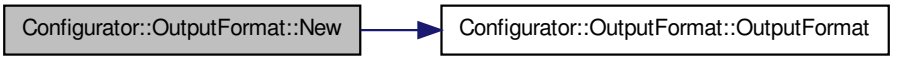
const ::std::string & Configurator::OutputFormat::name ( ) const [inline]

Here is the caller graph for this function:



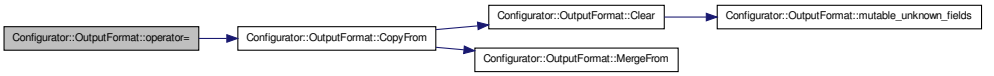
OutputFormat * Configurator::OutputFormat::New ( ) const

Here is the call graph for this function:



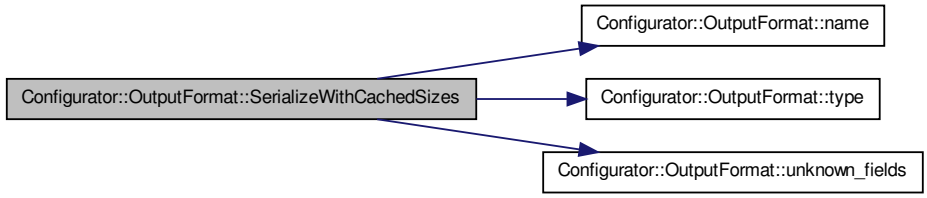
OutputFormat& Configurator::OutputFormat::operator= ([const OutputFormat &]from) [inline]

Here is the call graph for this function:



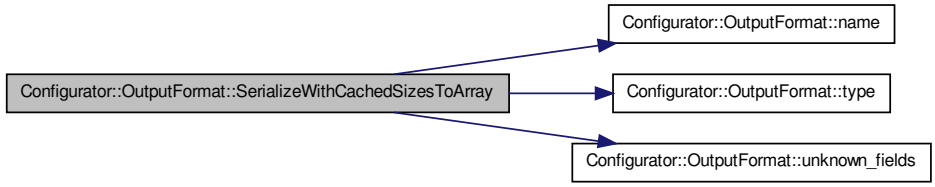
void Configurator::OutputFormat::SerializeWithCachedSizes ([::google::protobuf::io::CodedOutputStream *]output) const

Here is the call graph for this function:



google::protobuf::uint8 * Configurator::OutputFormat::SerializeWithCachedSizesToArray ([::google::protobuf::uint8 *]output) const

Here is the call graph for this function:



void Configurator::OutputFormat::set_name ([const ::std::string &]value) [inline]

Here is the caller graph for this function:

void Configurator::OutputFormat::set_name ([const char *]value) [inline]

void Configurator::OutputFormat::set_name ([const char *]value, size_t *size*) [inline]

void Configurator::OutputFormat::set_type ([const char *]value) [inline]

void Configurator::OutputFormat::set_type ([const char *]value, size_t *size*) [inline]

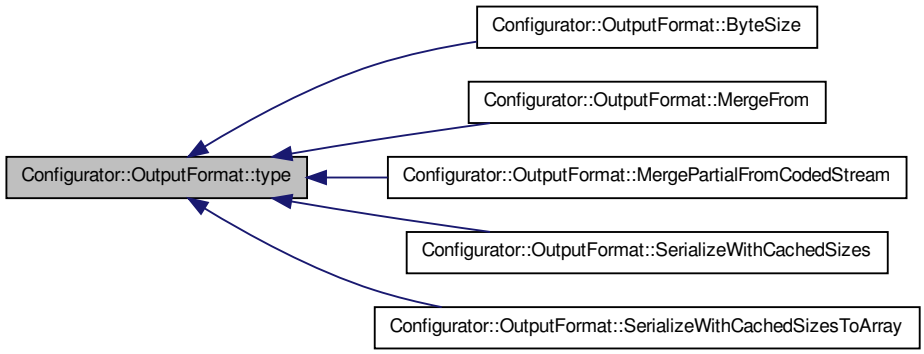
void Configurator::OutputFormat::set_type ([const ::std::string &]value) [inline]

Here is the caller graph for this function:

void Configurator::OutputFormat::Swap ([OutputFormat *]other)

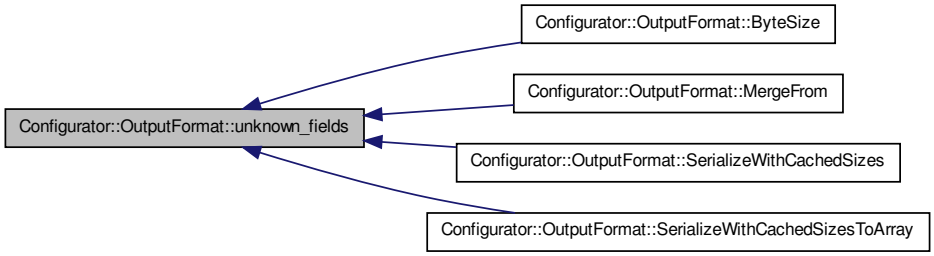
const ::std::string & Configurator::OutputFormat::type ( ) const [inline]

Here is the caller graph for this function:



const ::google::protobuf::UnknownFieldSet& Configurator::OutputFormat::unknown_-fields ( ) const [inline]

Here is the caller graph for this function:



7.51.7 Friends And Related Function Documentation

void protobuf_AddDesc_confproblem_2eproto ( ) [friend]

void protobuf_AssignDesc_confproblem_2eproto ( ) [friend]

void protobuf_ShutdownFile_confproblem_2eproto ( ) [friend]

7.51.8 Member Data Documentation

const int Configurator::OutputFormat::kNameFieldNumber = 1 `[static]`

const int Configurator::OutputFormat::kTypeFieldNumber = 2 `[static]`

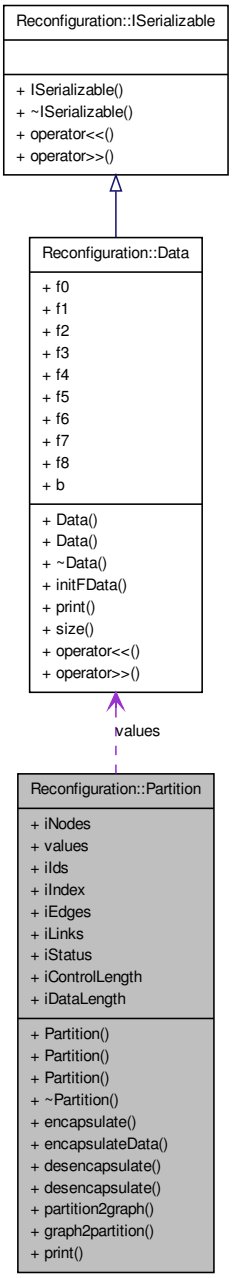
The documentation for this class was generated from the following files:

- confproblem.pb.h
- confproblem.pb.cc

7.52 Reconfiguration::Partition Class Reference

`#include <Problem.h>`

Collaboration diagram for Reconfiguration::Partition:



7.52.1 Public Member Functions

- Partition ()

  Constructor of the Partition class.

- Partition (int, int)

  Constructor of the MapElements class.

- Partition (const Partition &)

  Copy constructor of the Partition class.

- virtual ∼Partition ()

  virtual destructor of the class Partition.

- double ∗ encapsulate ()

  Encapsulates the partition into a double array.

- char ∗ encapsulateData ()

  Encapsulates the data array into a char array.

- void desencapsulate (double ∗)

  Desencapsulates a buffer into a partition without data.

- void desencapsulate (char ∗, int)

  Desencapsulates a char buffer into the data field of a partition.

- void partition2graph (Graph ∗)

  Creates a graph from the partition.

- void graph2partition (Graph ∗)

  Dumps a grapth into a partition.

- void print ()

  prints the Partition object.

7.52.2 Public Attributes

- int iNodes
- Data * values
- int * iIds
- int * iIndex
- int * iEdges
- int * iLinks
- int * iStatus
- int iControlLength
- int iDataLength

7.52.3 Detailed Description

Defines the partition to be sent to a certain slave. Allocates information about the number of nodes, its data, its identifiers, the indices, the edges identifiers, the links and the status of each vertex. The total length of the partition as the total length of date is also stored. The ith entry of array index stores the total number of neighbors of the first i graph nodes. Thus, index[0] is the degree of node zero and index[i]-index[i-1] is the degree of node i.

7.52.4 Constructor & Destructor Documentation

Reconfiguration::Partition::Partition ( )

Constructor of the Partition class.

Returns

*this

Creates an object Partition. Initiates all arrays to NULL.

Reconfiguration::Partition::Partition ([int]iNodes, int *iLink*)

Constructor of the MapElements class.

Parameters

in	*iNodes*	Number of nodes of the partition.
in	*iLink*	Number of links of the partition.

Returns

∗this

Creates an object MapElements. Allocates the arrays.

Reconfiguration::Partition::Partition ([const Partition &]partition)

Copy constructor of the Partition class.

Parameters

in	*machine*	Object to be copied.

Returns

∗this

Copies an object Partition.

Reconfiguration::Partition::∼Partition ( ) [virtual]

virtual destructor of the class Partition.

Returns

void

Destroys the Partition object.

7.52.5 Member Function Documentation

void Reconfiguration::Partition::desencapsulate ([double ∗]buffer)

Desencapsulates a buffer into a partition without data.

Parameters

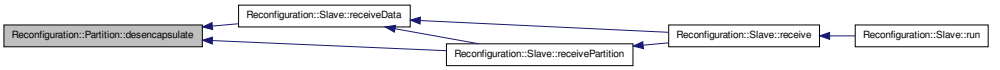
in	*buffer*	Buffer to desencapsulate.

Returns

void

Fills the new partition desencapsulating the information of the buffer.

Here is the caller graph for this function:



void Reconfiguration::Partition::desencapsulate ([char ∗]buffer, int *iSize*)

Desencapsulates a char buffer into the data field of a partition.

Parameters

in	*buffer*	Buffer to be desencapsulated.
in	*iSize*	Buffer length.

Returns

void

Fills the data field of the partition desencapsulating the buffer.

double ∗ Reconfiguration::Partition::encapsulate ( )

Encapsulates the partition into a double array.

Returns

double∗

Encapsulates the partition into a double array. Sets the iControlLength field to its correspondent value.

char ∗ Reconfiguration::Partition::encapsulateData ( )

Encapsulates the data array into a char array.

Returns

char∗

Encapsulates the data into a char array, calling the data serializing method. Sets the iDataLength field to its correspondent value.

Here is the caller graph for this function:

void Reconfiguration::Partition::graph2partition ([Graph *]graph)

Dumps a grapth into a partition.

Parameters

in	*graph*	Graph to be dumped.

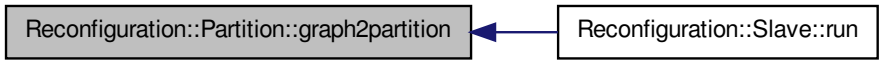
Returns

void

Fills a partition from the information of the graph.

The memory for the partition must be allocated before this function is called.

Here is the caller graph for this function:



void Reconfiguration::Partition::partition2graph ([Graph *]graph)

Creates a graph from the partition.

Parameters

out	*graph*	Graph created with the partition information.

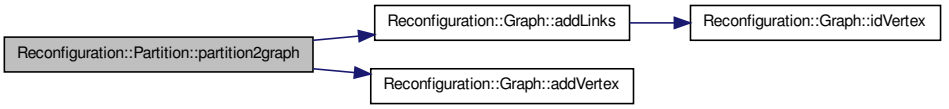
Returns

void

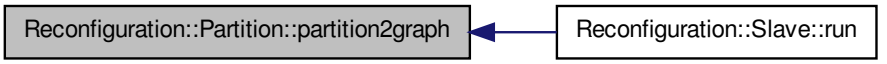
Creates a graph from the information of the partition.

The memory for the graph must be allocated before this function is called.

Here is the call graph for this function:



Here is the caller graph for this function:



void Reconfiguration::Partition::print ( )

prints the Partition object.

Returns

void

Prints the Partition object.

7.52.6 Member Data Documentation

int Reconfiguration::Partition::iControlLength

Length (ints) of the dataframe (without data values):

iNodes: 1

iIds, iIndex, iStatus: as much as nodes.

iLinks, iEdges: as much as iIndex[iNodes-1].

int Reconfiguration::Partition::iDataLength

Buffer length of serialized datas.

int * Reconfiguration::Partition::iEdges

Identifiers of the node neighbors.

int * Reconfiguration::Partition::iIds

Nodes identifiers.

int * Reconfiguration::Partition::iIndex

Node indices. The ith entry of array index stores the total number of neighbors of the first i graph nodes. Thus, index[0] is the degree of node zero and index[i]-index[i-1] is the degree of node i.

int * Reconfiguration::Partition::iLinks

Weights of each link per neighbor node.

int Reconfiguration::Partition::iNodes

Number of nodes at the partition.

int * Reconfiguration::Partition::iStatus

Status of each node.

See also

UPDATE

Data * Reconfiguration::Partition::values

Nodes values.

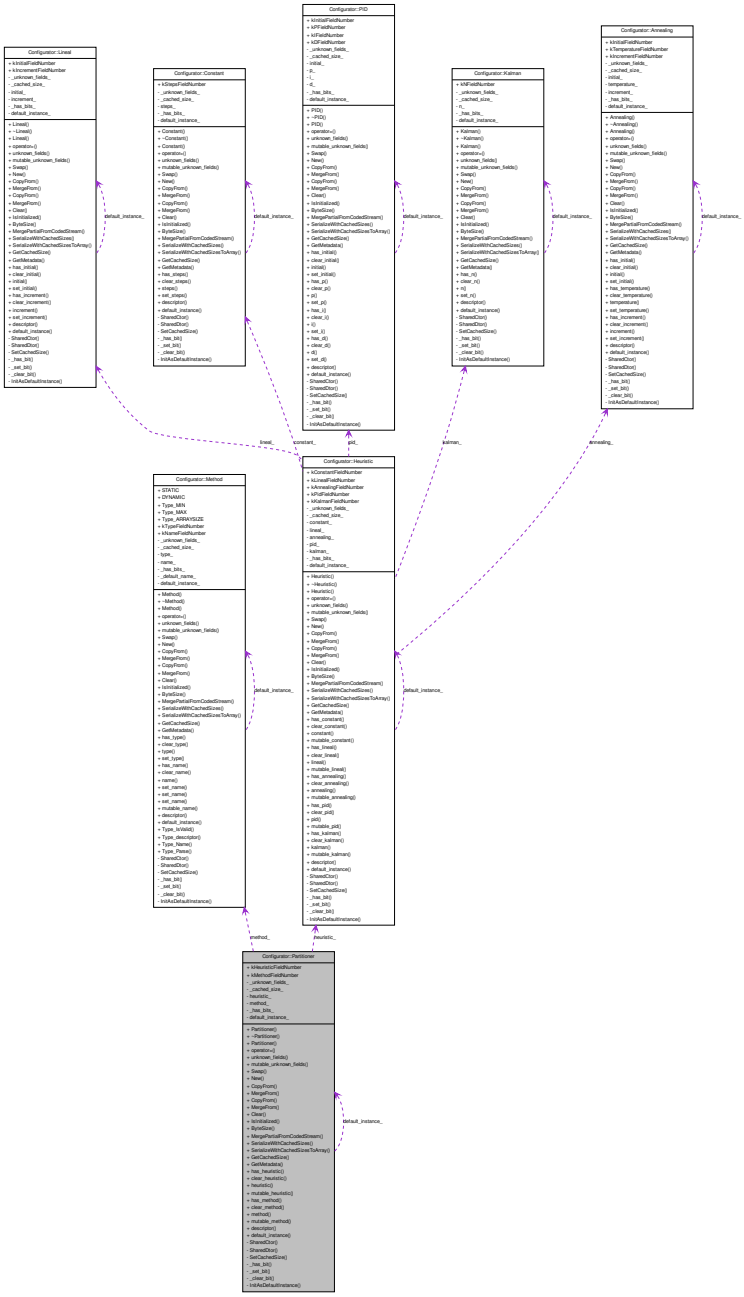
The documentation for this class was generated from the following files:

- Problem.h
- Problem.cpp

7.53 Configurator::Partitioner Class Reference

`#include <confpartitioner.pb.h>`

Collaboration diagram for Configurator::Partitioner:



7.53.1 Public Member Functions

- Partitioner ()
- virtual ∼Partitioner ()
- Partitioner (const Partitioner &from)
- Partitioner & operator= (const Partitioner &from)
- const ::google::protobuf::UnknownFieldSet & unknown_fields () const
- inline::google::protobuf::UnknownFieldSet ∗ mutable_unknown_fields ()
- void Swap (Partitioner ∗other)
- Partitioner ∗ New () const
- void CopyFrom (const ::google::protobuf::Message &from)
- void MergeFrom (const ::google::protobuf::Message &from)
- void CopyFrom (const Partitioner &from)
- void MergeFrom (const Partitioner &from)
- void Clear ()
- bool IsInitialized () const
- int ByteSize () const
- bool MergePartialFromCodedStream (::google::protobuf::io::CodedInputStream ∗input)
- void SerializeWithCachedSizes (::google::protobuf::io::CodedOutputStream ∗output) const
- ::google::protobuf::uint8 ∗ SerializeWithCachedSizesToArray (::google::protobuf::uint8 ∗output) const
- int GetCachedSize () const
- ::google::protobuf::Metadata GetMetadata () const
- bool has_heuristic () const
- void clear_heuristic ()
- const ::Configurator::Heuristic & heuristic () const
- inline::Configurator::Heuristic ∗ mutable_heuristic ()
- bool has_method () const
- void clear_method ()
- const ::Configurator::Method & method () const
- inline::Configurator::Method ∗ mutable_method ()

7.53.2 Static Public Member Functions

- static const ::google::protobuf::Descriptor * descriptor ()
- static const Partitioner & default_instance ()

7.53.3 Static Public Attributes

- static const int kHeuristicFieldNumber = 1
- static const int kMethodFieldNumber = 2

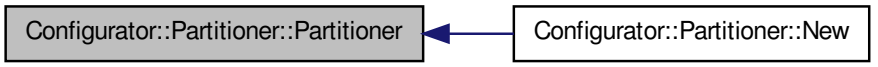
7.53.4 Friends

- void protobuf_AddDesc_confpartitioner_2eproto ()
- void protobuf_AssignDesc_confpartitioner_2eproto ()
- void protobuf_ShutdownFile_confpartitioner_2eproto ()

7.53.5 Constructor & Destructor Documentation

Configurator::Partitioner::Partitioner ( )

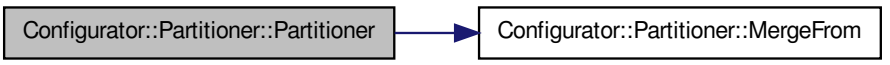
Here is the caller graph for this function:



Configurator::Partitioner::~Partitioner ( ) [virtual]

Configurator::Partitioner::Partitioner ([const Partitioner &]from)

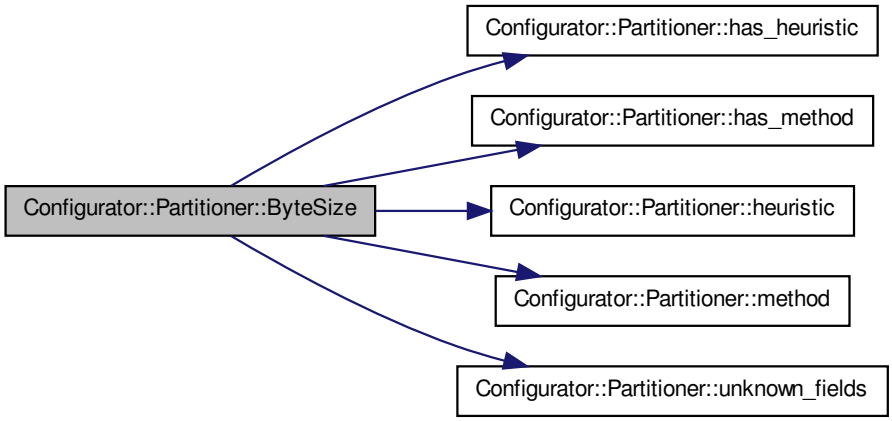
Here is the call graph for this function:



7.53.6 Member Function Documentation

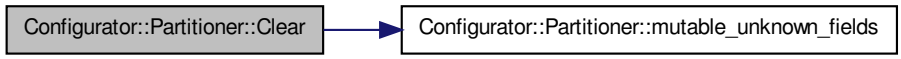
int Configurator::Partitioner::ByteSize ( ) const

Here is the call graph for this function:

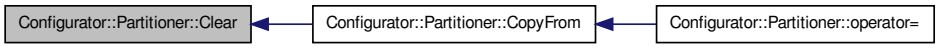


void Configurator::Partitioner::Clear ( )

Here is the call graph for this function:



Here is the caller graph for this function:

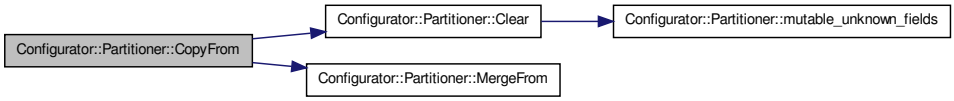


void Configurator::Partitioner::clear_heuristic ( ) [inline]

void Configurator::Partitioner::clear_method ( ) [inline]

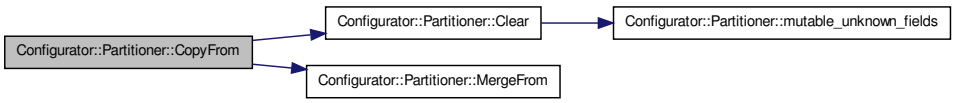
void Configurator::Partitioner::CopyFrom ([const Partitioner &]from)

Here is the call graph for this function:



void Configurator::Partitioner::CopyFrom ([const ::google::protobuf::Message &]from)

Here is the call graph for this function:



Here is the caller graph for this function:

const Partitioner & Configurator::Partitioner::default_instance ( ) [static]

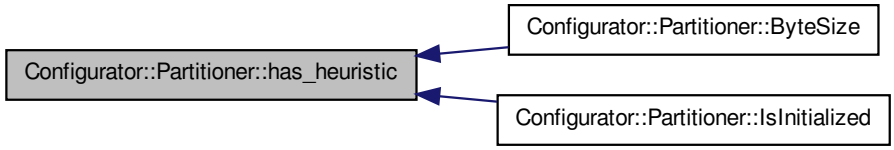
const ::google::protobuf::Descriptor * Configurator::Partitioner::descriptor ( ) [static]

int Configurator::Partitioner::GetCachedSize ( ) const [inline]

google::protobuf::Metadata Configurator::Partitioner::GetMetadata ( ) const

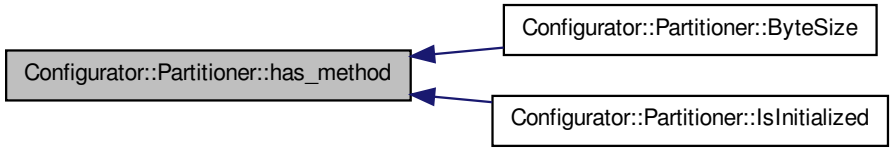
bool Configurator::Partitioner::has_heuristic ( ) const [inline]

Here is the caller graph for this function:



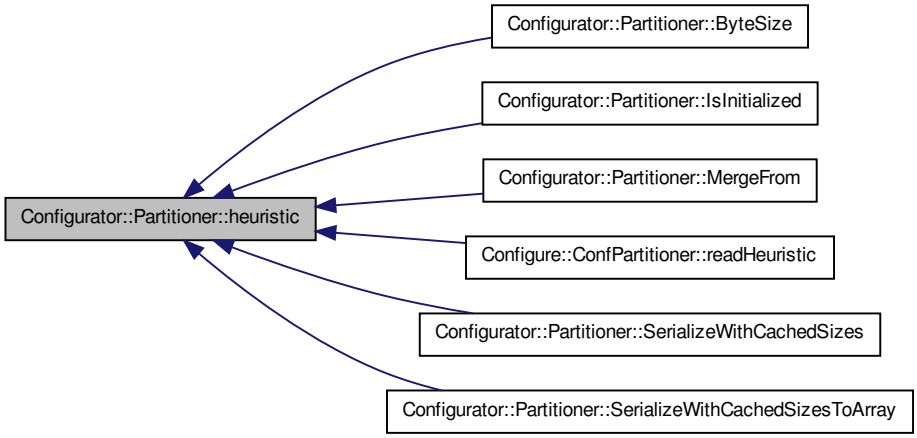
bool Configurator::Partitioner::has_method ( ) const [inline]

Here is the caller graph for this function:



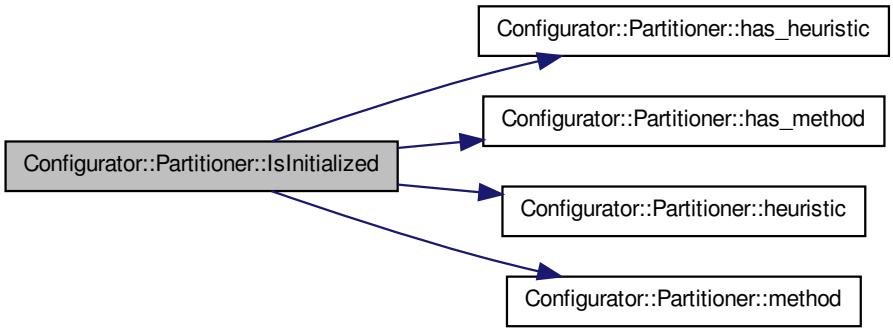
const ::Configurator::Heuristic & Configurator::Partitioner::heuristic ( ) const [inline]

Here is the caller graph for this function:



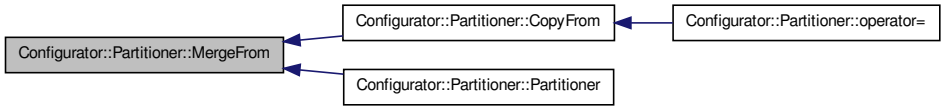
bool Configurator::Partitioner::IsInitialized ( ) const

Here is the call graph for this function:



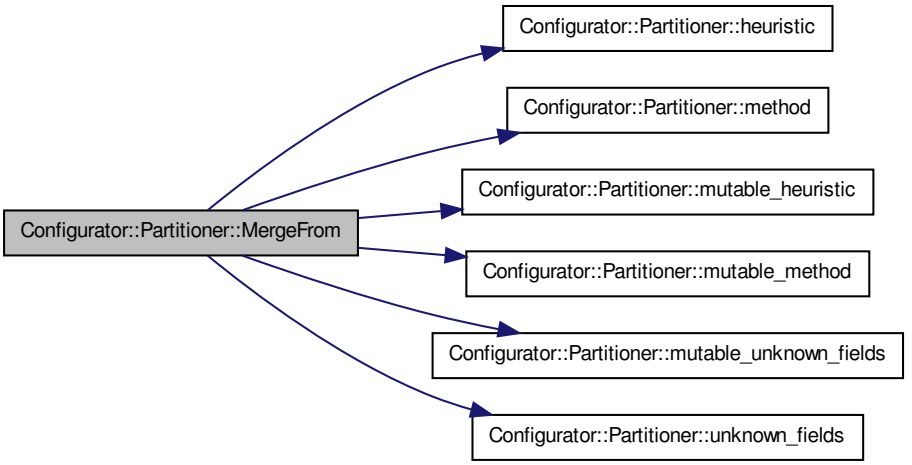
void Configurator::Partitioner::MergeFrom ([const ::google::protobuf::Message &]from)

Here is the caller graph for this function:



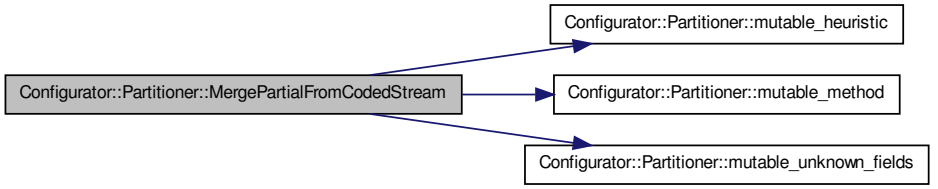
void Configurator::Partitioner::MergeFrom ([const Partitioner &]from)

Here is the call graph for this function:



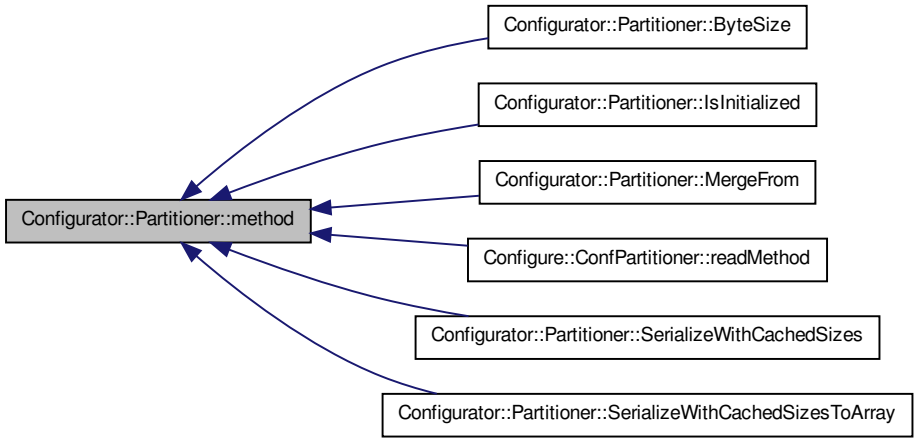
bool Configurator::Partitioner::MergePartialFromCodedStream ([::google::protobuf::io::CodedInputStream *]input)

Here is the call graph for this function:



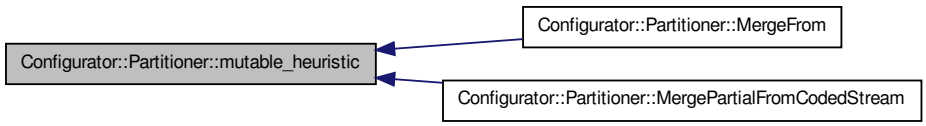
const ::Configurator::Method & Configurator::Partitioner::method ( ) const `[inline]`

Here is the caller graph for this function:



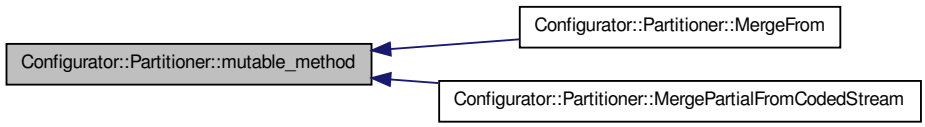
Configurator::Heuristic * Configurator::Partitioner::mutable_heuristic ( ) [inline]

Here is the caller graph for this function:



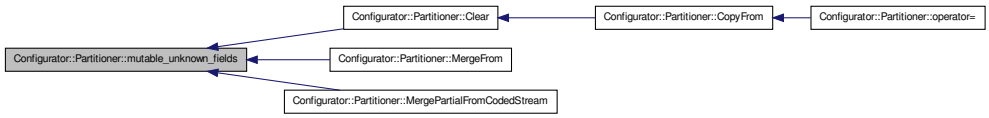
Configurator::Method * Configurator::Partitioner::mutable_method ( ) [inline]

Here is the caller graph for this function:



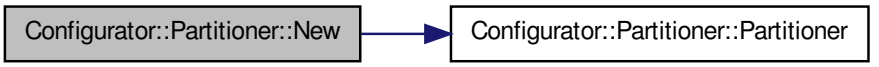
inline ::google::protobuf::UnknownFieldSet* Configurator::Partitioner::mutable_-unknown_fields ( ) [inline]

Here is the caller graph for this function:



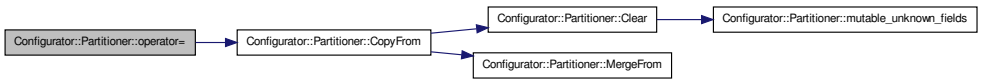
Partitioner * Configurator::Partitioner::New ( ) const

Here is the call graph for this function:



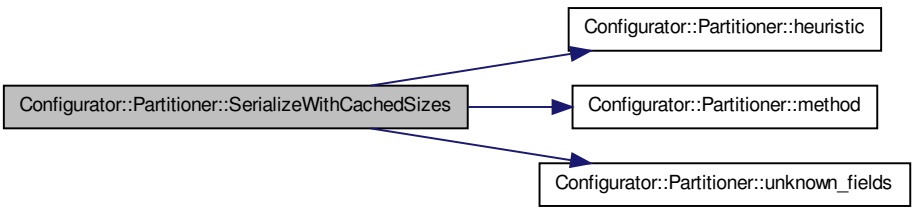
Partitioner& Configurator::Partitioner::operator= ([const Partitioner &]from) `[inline]`

Here is the call graph for this function:



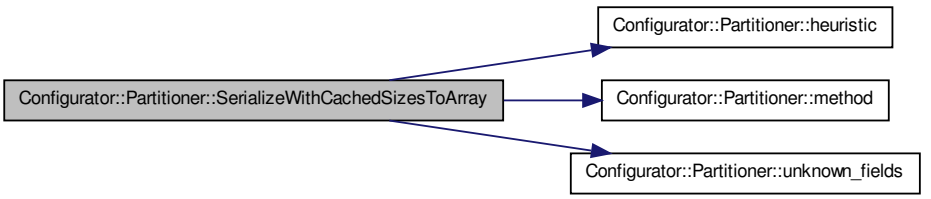
void Configurator::Partitioner::SerializeWithCachedSizes ([::google::protobuf::io::CodedOutputStream ∗]output) const

Here is the call graph for this function:



google::protobuf::uint8 ∗ Configurator::Partitioner::SerializeWithCachedSizesToArray ([::google::protobuf::uint8 ∗]output) const

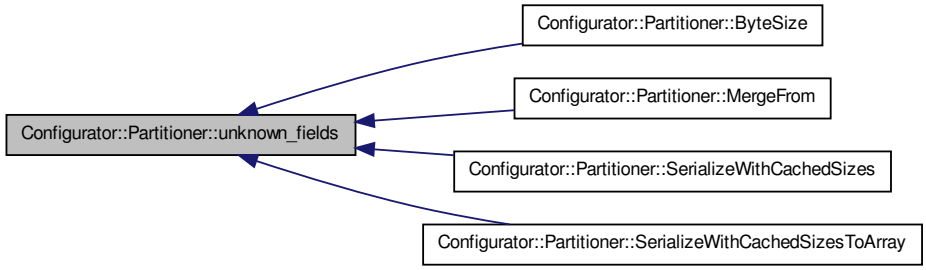
Here is the call graph for this function:



void Configurator::Partitioner::Swap ([Partitioner ∗]other)

const ::google::protobuf::UnknownFieldSet& Configurator::Partitioner::unknown_-fields ( ) const `[inline]`

Here is the caller graph for this function:



7.53.7 Friends And Related Function Documentation

void protobuf_AddDesc_confpartitioner_2eproto ( ) [friend]

void protobuf_AssignDesc_confpartitioner_2eproto ( ) [friend]

void protobuf_ShutdownFile_confpartitioner_2eproto ( ) [friend]

7.53.8 Member Data Documentation

const int Configurator::Partitioner::kHeuristicFieldNumber = 1 [static]

const int Configurator::Partitioner::kMethodFieldNumber = 2 [static]

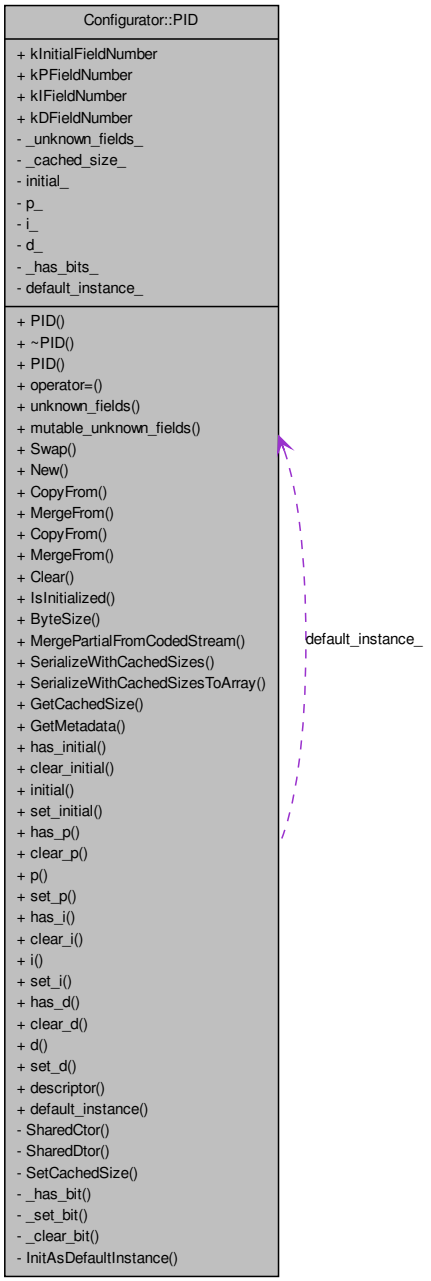
The documentation for this class was generated from the following files:

- confpartitioner.pb.h
- confpartitioner.pb.cc

7.54 Configurator::PID Class Reference

`#include <confpartitioner.pb.h>`

Collaboration diagram for Configurator::PID:



7.54.1 Public Member Functions

- PID ()
- virtual ∼PID ()
- PID (const PID &from)
- PID & operator= (const PID &from)
- const ::google::protobuf::UnknownFieldSet & unknown_fields () const
- inline::google::protobuf::UnknownFieldSet ∗ mutable_unknown_fields ()
- void Swap (PID ∗other)
- PID ∗ New () const
- void CopyFrom (const ::google::protobuf::Message &from)
- void MergeFrom (const ::google::protobuf::Message &from)
- void CopyFrom (const PID &from)
- void MergeFrom (const PID &from)
- void Clear ()
- bool IsInitialized () const
- int ByteSize () const
- bool MergePartialFromCodedStream (::google::protobuf::io::CodedInputStream ∗input)
- void SerializeWithCachedSizes (::google::protobuf::io::CodedOutputStream ∗output) const
- ::google::protobuf::uint8 ∗ SerializeWithCachedSizesToArray (::google::protobuf::uint8 ∗output) const
- int GetCachedSize () const
- ::google::protobuf::Metadata GetMetadata () const
- bool has_initial () const
- void clear_initial ()
- inline::google::protobuf::int32 initial () const
- void set_initial (::google::protobuf::int32 value)
- bool has_p () const
- void clear_p ()
- double p () const
- void set_p (double value)
- bool has_i () const
- void clear_i ()

- double i () const
- void set_i (double value)
- bool has_d () const
- void clear_d ()
- double d () const
- void set_d (double value)

7.54.2 Static Public Member Functions

- static const ::google::protobuf::Descriptor * descriptor ()
- static const PID & default_instance ()

7.54.3 Static Public Attributes

- static const int kInitialFieldNumber = 1
- static const int kPFieldNumber = 2
- static const int kIFieldNumber = 3
- static const int kDFieldNumber = 4

7.54.4 Friends

- void protobuf_AddDesc_confpartitioner_2eproto ()
- void protobuf_AssignDesc_confpartitioner_2eproto ()
- void protobuf_ShutdownFile_confpartitioner_2eproto ()

7.54.5 Constructor & Destructor Documentation

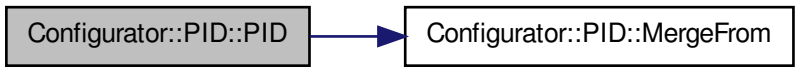
Configurator::PID::PID ( )

Here is the caller graph for this function:

Configurator::PID::∼PID ( ) [virtual]

Configurator::PID::PID ([const PID &]from)

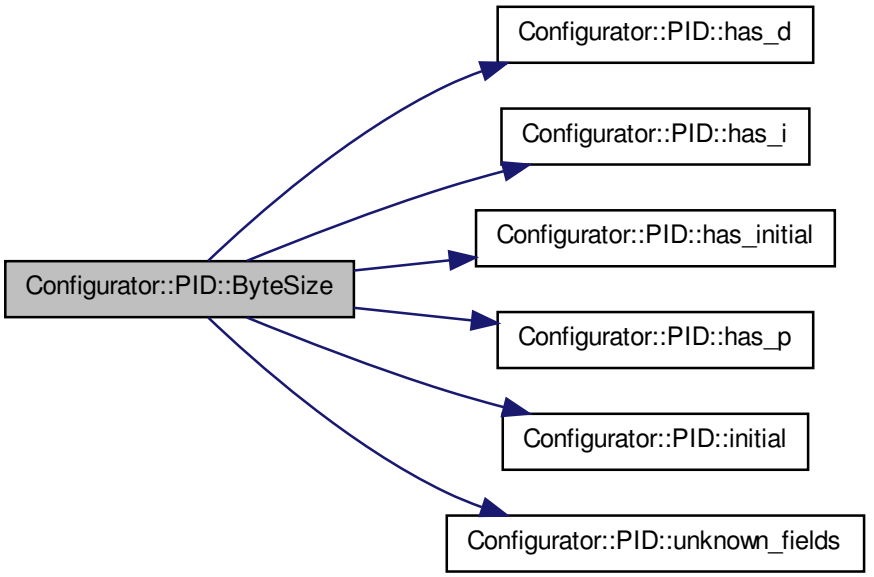
Here is the call graph for this function:



7.54.6 Member Function Documentation

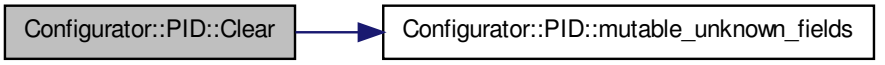
int Configurator::PID::ByteSize ( ) const

Here is the call graph for this function:

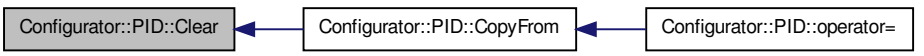


void Configurator::PID::Clear ( )

Here is the call graph for this function:



Here is the caller graph for this function:



void Configurator::PID::clear_d ( ) [inline]

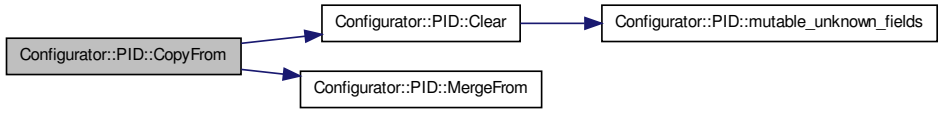
void Configurator::PID::clear_i ( ) [inline]

void Configurator::PID::clear_initial ( ) [inline]

void Configurator::PID::clear_p ( ) [inline]

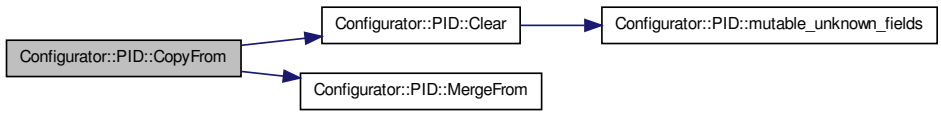
void Configurator::PID::CopyFrom ([const PID &]from)

Here is the call graph for this function:



void Configurator::PID::CopyFrom ([const ::google::protobuf::Message &]from)

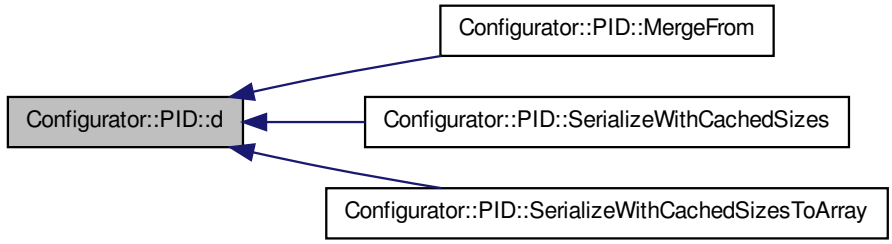
Here is the call graph for this function:



Here is the caller graph for this function:

double Configurator::PID::d ( ) const [inline]

Here is the caller graph for this function:



const PID & Configurator::PID::default_instance ( ) [static]

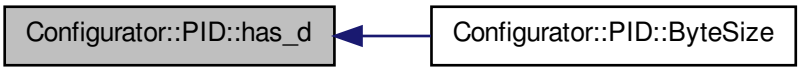
const ::google::protobuf::Descriptor * Configurator::PID::descriptor ( ) [static]

int Configurator::PID::GetCachedSize ( ) const [inline]

google::protobuf::Metadata Configurator::PID::GetMetadata ( ) const

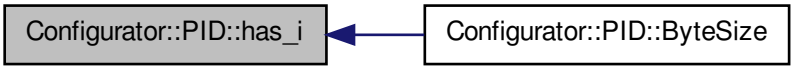
bool Configurator::PID::has_d ( ) const [inline]

Here is the caller graph for this function:



bool Configurator::PID::has_i ( ) const [inline]

Here is the caller graph for this function:

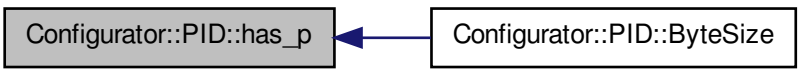


bool Configurator::PID::has_initial ( ) const [inline]

Here is the caller graph for this function:

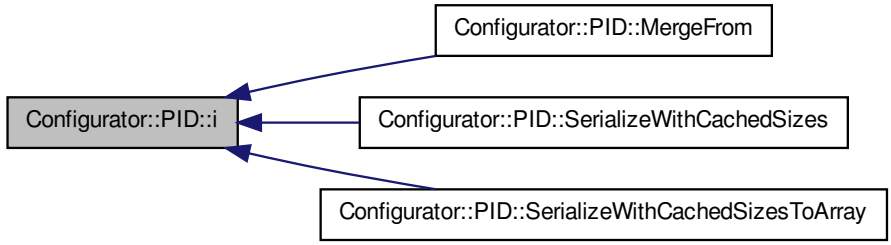
bool Configurator::PID::has_p ( ) const [inline]

Here is the caller graph for this function:



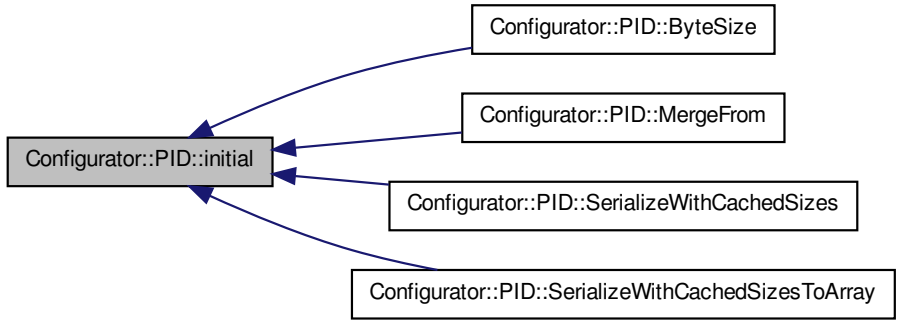
double Configurator::PID::i ( ) const [inline]

Here is the caller graph for this function:



google::protobuf::int32 Configurator::PID::initial ( ) const [inline]

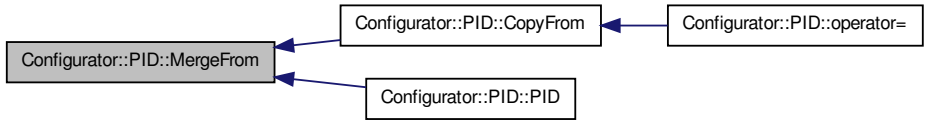
Here is the caller graph for this function:



bool Configurator::PID::IsInitialized ( ) const

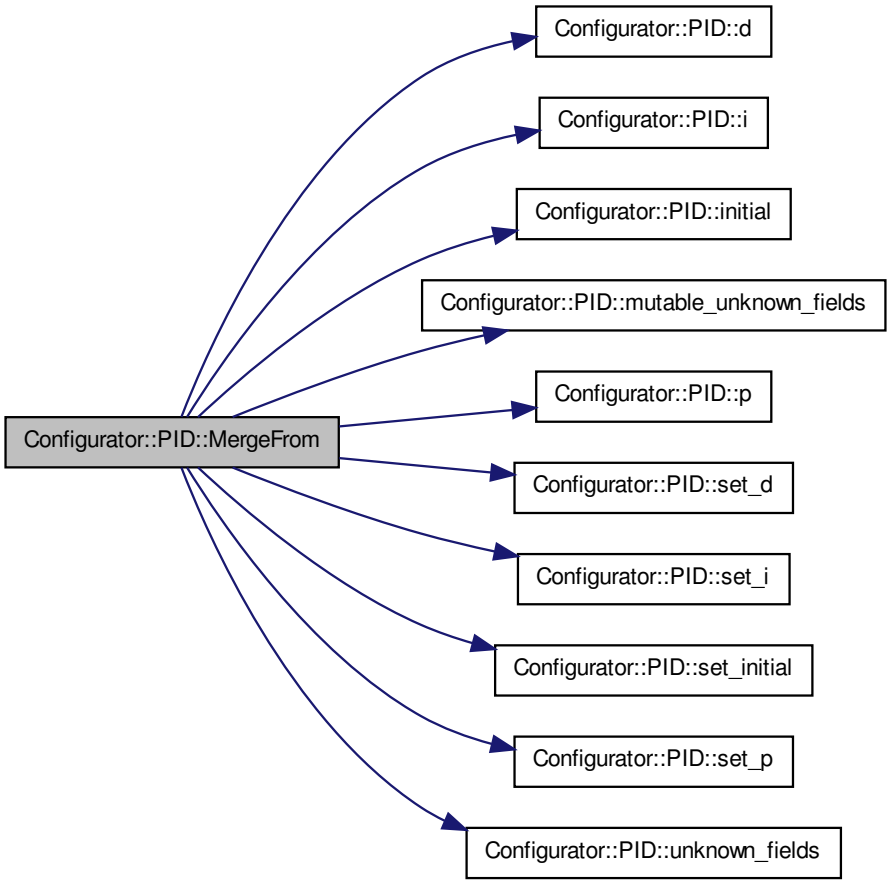
void Configurator::PID::MergeFrom ([const ::google::protobuf::Message &]from)

Here is the caller graph for this function:



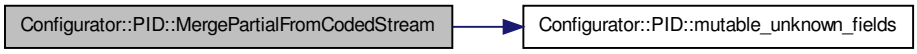
void Configurator::PID::MergeFrom ([const PID &]from)

Here is the call graph for this function:



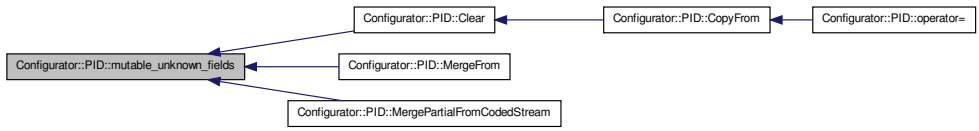
bool Configurator::PID::MergePartialFromCodedStream ([::google::protobuf::io::CodedInputStream ∗]input)

Here is the call graph for this function:



inline ::google::protobuf::UnknownFieldSet∗ Configurator::PID::mutable_unknown_-fields ( ) [inline]

Here is the caller graph for this function:

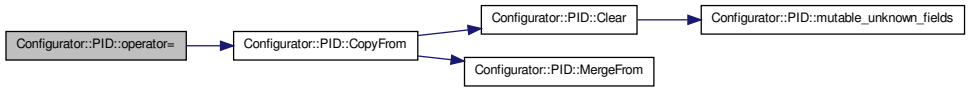


PID * Configurator::PID::New ( ) const

Here is the call graph for this function:

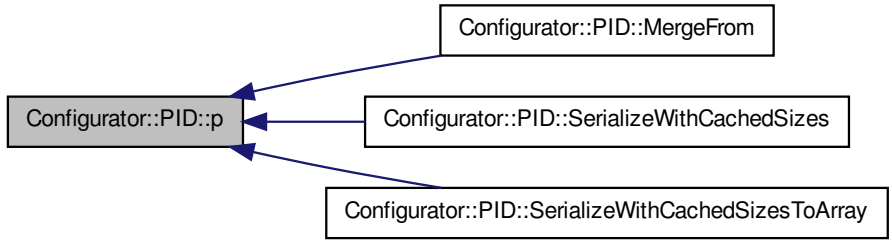
PID& Configurator::PID::operator= ([const PID &]from) `[inline]`

Here is the call graph for this function:



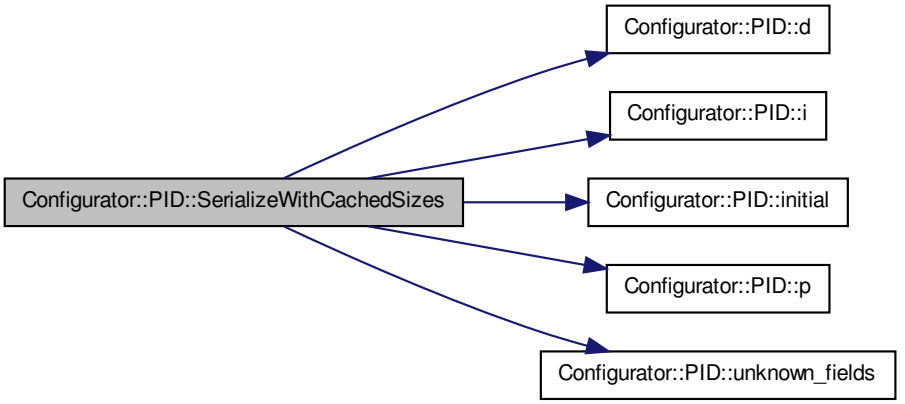
double Configurator::PID::p ( ) const `[inline]`

Here is the caller graph for this function:



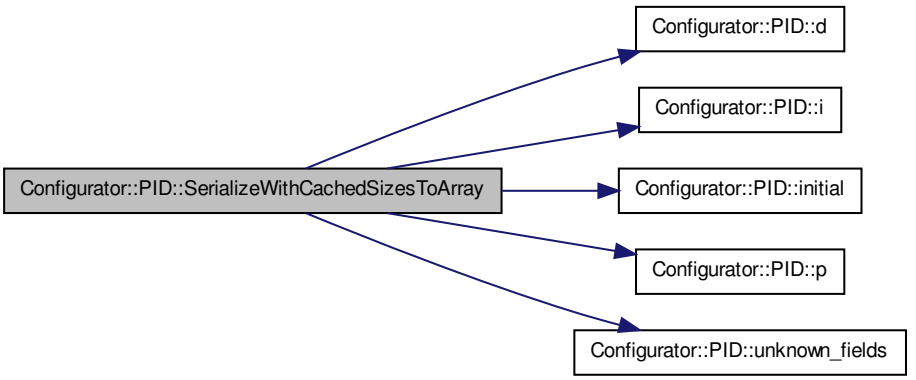
void Configurator::PID::SerializeWithCachedSizes ([::google::protobuf::io::CodedOutputStream *]output) const

Here is the call graph for this function:



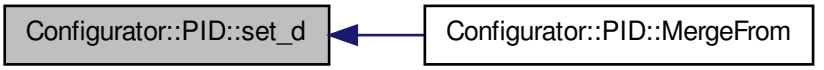
google::protobuf::uint8 * Configurator::PID::SerializeWithCachedSizesToArray ([::google::protobuf::uint8 *]output) const

Here is the call graph for this function:



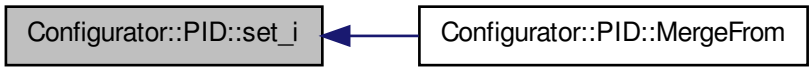
void Configurator::PID::set_d ([double]value) `[inline]`

Here is the caller graph for this function:



void Configurator::PID::set_i ([double]value) `[inline]`

Here is the caller graph for this function:

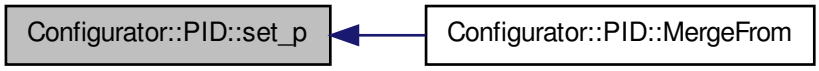


void Configurator::PID::set_initial ([::google::protobuf::int32]value) `[inline]`

Here is the caller graph for this function:

void Configurator::PID::set_p ([double]value) `[inline]`

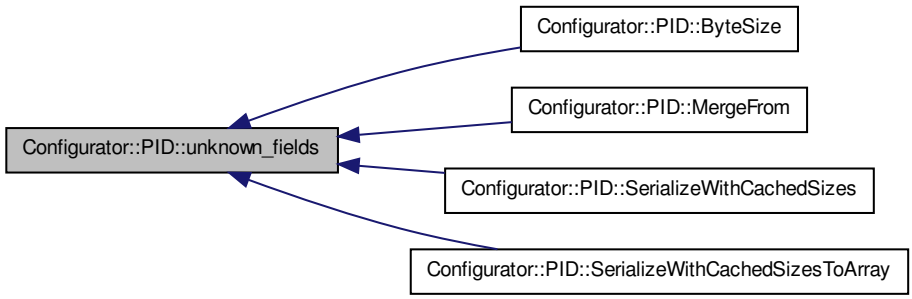
Here is the caller graph for this function:



void Configurator::PID::Swap ([PID *]other)

const ::google::protobuf::UnknownFieldSet& Configurator::PID::unknown_fields ( ) const `[inline]`

Here is the caller graph for this function:



7.54.7 Friends And Related Function Documentation

void protobuf_AddDesc_confpartitioner_2eproto ( ) [friend]

void protobuf_AssignDesc_confpartitioner_2eproto ( ) [friend]

void protobuf_ShutdownFile_confpartitioner_2eproto ( ) [friend]

7.54.8 Member Data Documentation

const int Configurator::PID::kDFieldNumber = 4 [static]

const int Configurator::PID::kIFieldNumber = 3 [static]

const int Configurator::PID::kInitialFieldNumber = 1 [static]

const int Configurator::PID::kPFieldNumber = 2 [static]

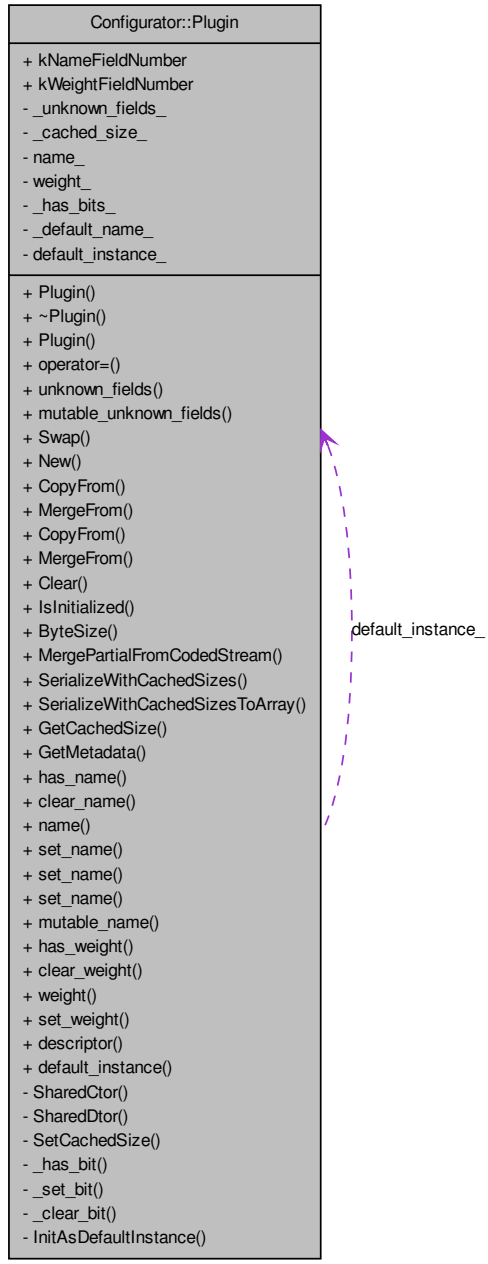
The documentation for this class was generated from the following files:

- confpartitioner.pb.h
- confpartitioner.pb.cc

7.55 Configurator::Plugin Class Reference

`#include <confloadbalancer.pb.h>`

Collaboration diagram for Configurator::Plugin:



7.55.1 Public Member Functions

- Plugin ()
- virtual ~Plugin ()
- Plugin (const Plugin &from)
- Plugin & operator= (const Plugin &from)
- const ::google::protobuf::UnknownFieldSet & unknown_fields () const
- inline::google::protobuf::UnknownFieldSet * mutable_unknown_fields ()
- void Swap (Plugin *other)
- Plugin * New () const
- void CopyFrom (const ::google::protobuf::Message &from)
- void MergeFrom (const ::google::protobuf::Message &from)
- void CopyFrom (const Plugin &from)
- void MergeFrom (const Plugin &from)
- void Clear ()
- bool IsInitialized () const
- int ByteSize () const
- bool MergePartialFromCodedStream (::google::protobuf::io::CodedInputStream *input)
- void SerializeWithCachedSizes (::google::protobuf::io::CodedOutputStream *output) const
- ::google::protobuf::uint8 * SerializeWithCachedSizesToArray (::google::protobuf::uint8 *output) const
- int GetCachedSize () const
- ::google::protobuf::Metadata GetMetadata () const
- bool has_name () const
- void clear_name ()
- const ::std::string & name () const
- void set_name (const ::std::string &value)
- void set_name (const char *value)
- void set_name (const char *value, size_t size)
- inline::std::string * mutable_name ()
- bool has_weight () const
- void clear_weight ()
- double weight () const

- void set_weight (double value)

7.55.2 Static Public Member Functions

- static const ::google::protobuf::Descriptor * descriptor ()
- static const Plugin & default_instance ()

7.55.3 Static Public Attributes

- static const int kNameFieldNumber = 1
- static const int kWeightFieldNumber = 2

7.55.4 Friends

- void protobuf_AddDesc_confloadbalancer_2eproto ()
- void protobuf_AssignDesc_confloadbalancer_2eproto ()
- void protobuf_ShutdownFile_confloadbalancer_2eproto ()

7.55.5 Constructor & Destructor Documentation

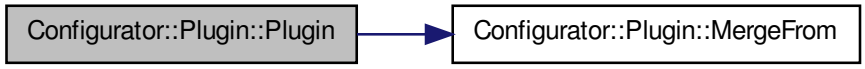
Configurator::Plugin::Plugin ( )

Here is the caller graph for this function:

Configurator::Plugin::~Plugin ( ) [virtual]

Configurator::Plugin::Plugin ([const Plugin &]from)

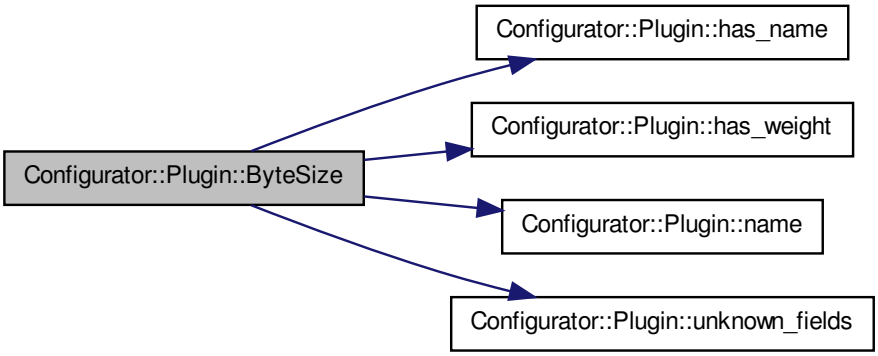
Here is the call graph for this function:



7.55.6 Member Function Documentation

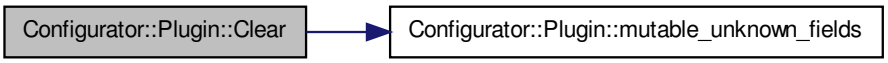
int Configurator::Plugin::ByteSize ( ) const

Here is the call graph for this function:

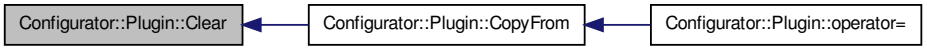


void Configurator::Plugin::Clear ( )

Here is the call graph for this function:



Here is the caller graph for this function:

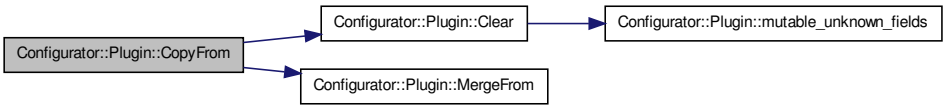


void Configurator::Plugin::clear_name ( ) [inline]

void Configurator::Plugin::clear_weight ( ) [inline]

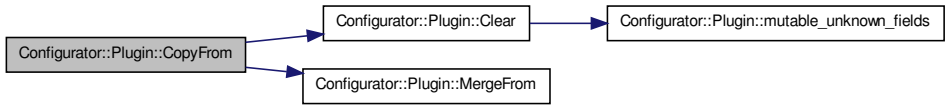
void Configurator::Plugin::CopyFrom ([const Plugin &]from)

Here is the call graph for this function:



void Configurator::Plugin::CopyFrom ([const ::google::protobuf::Message &]from)

Here is the call graph for this function:



Here is the caller graph for this function:

const Plugin & Configurator::Plugin::default_instance ( ) [static]

const ::google::protobuf::Descriptor * Configurator::Plugin::descriptor ( ) [static]

int Configurator::Plugin::GetCachedSize ( ) const [inline]

google::protobuf::Metadata Configurator::Plugin::GetMetadata ( ) const

bool Configurator::Plugin::has_name ( ) const [inline]

Here is the caller graph for this function:

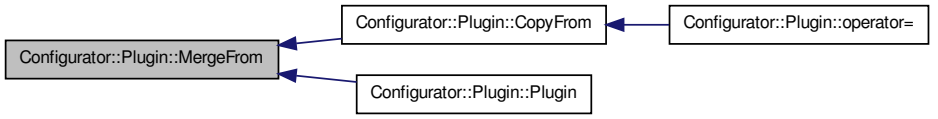
bool Configurator::Plugin::has_weight ( ) const [inline]

Here is the caller graph for this function:

bool Configurator::Plugin::IsInitialized ( ) const

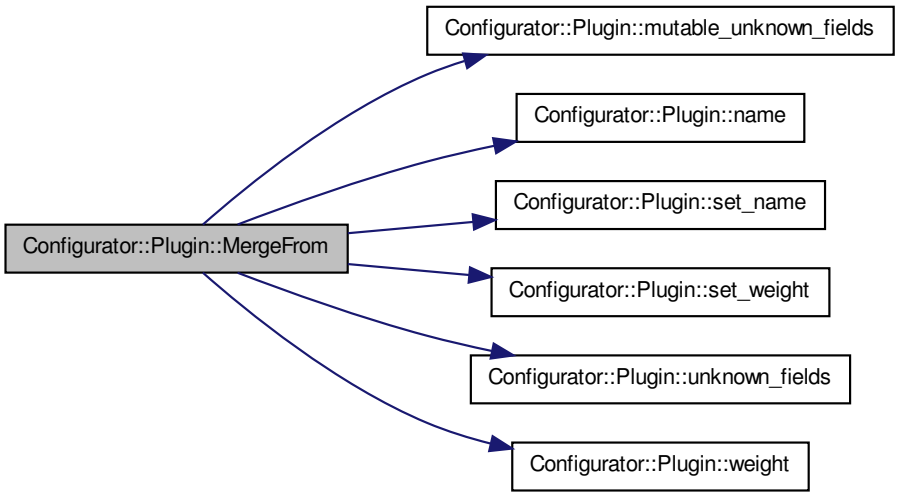
void Configurator::Plugin::MergeFrom ([const ::google::protobuf::Message &]from)

Here is the caller graph for this function:



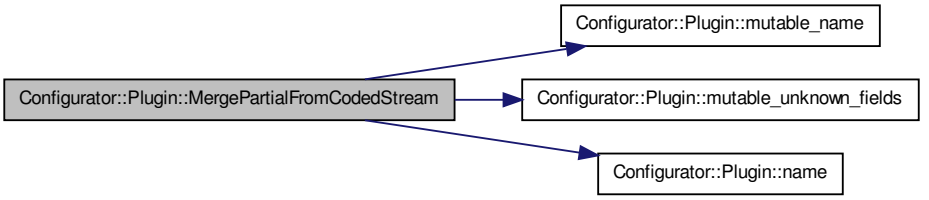
void Configurator::Plugin::MergeFrom ([const Plugin &]from)

Here is the call graph for this function:



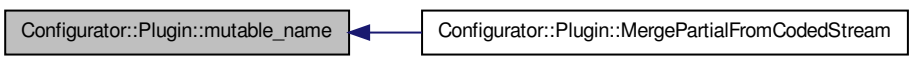
bool Configurator::Plugin::MergePartialFromCodedStream ([::google::protobuf::io::CodedInputStream *]input)

Here is the call graph for this function:



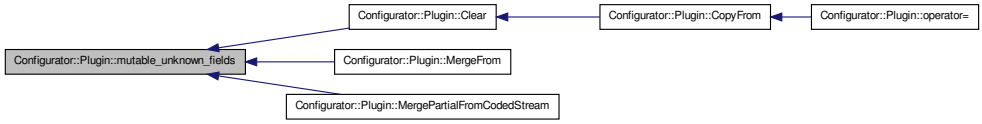
std::string * Configurator::Plugin::mutable_name ( ) `[inline]`

Here is the caller graph for this function:



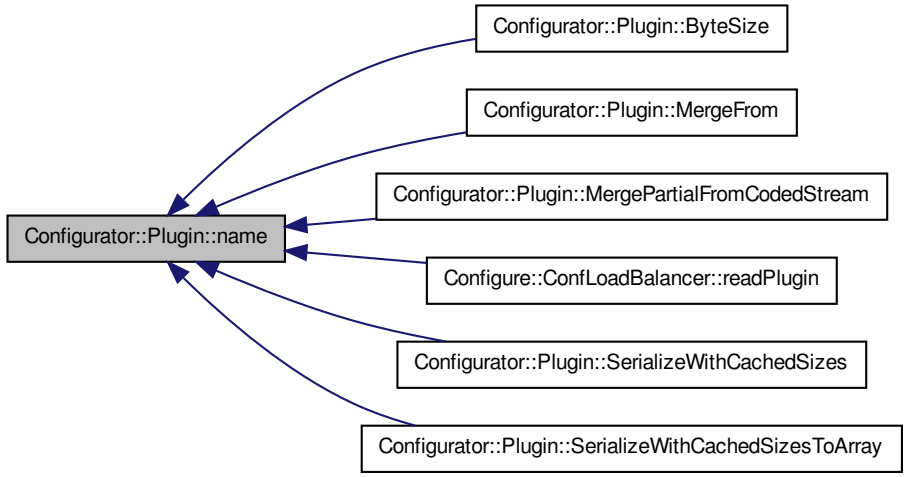
inline ::google::protobuf::UnknownFieldSet* Configurator::Plugin::mutable_-unknown_fields ( ) [inline]

Here is the caller graph for this function:



const ::std::string & Configurator::Plugin::name ( ) const [inline]

Here is the caller graph for this function:

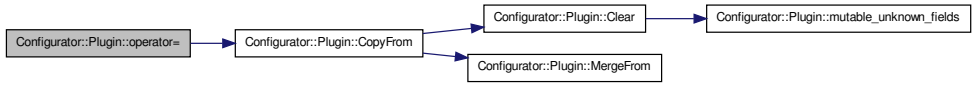


Plugin * Configurator::Plugin::New ( ) const

Here is the call graph for this function:

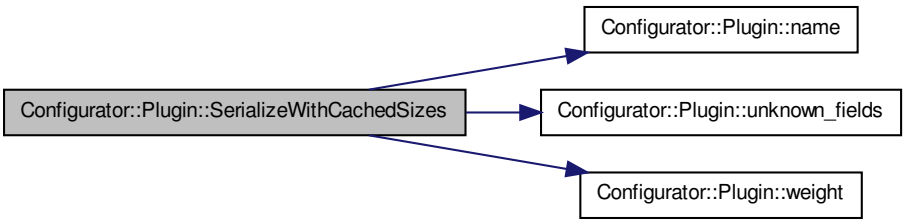
Plugin& Configurator::Plugin::operator= ([const Plugin &]from) [inline]

Here is the call graph for this function:



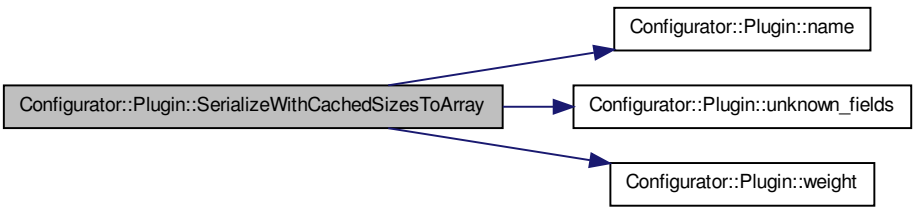
void Configurator::Plugin::SerializeWithCachedSizes ([::google::protobuf::io::CodedOutputStream *]output) const

Here is the call graph for this function:



google::protobuf::uint8 * Configurator::Plugin::SerializeWithCachedSizesToArray ([::google::protobuf::uint8 *]output) const

Here is the call graph for this function:



void Configurator::Plugin::set_name ([const ::std::string &]value) [inline]

Here is the caller graph for this function:

void Configurator::Plugin::set_name ([const char *]value) [inline]

void Configurator::Plugin::set_name ([const char *]value, size_t *size*) [inline]

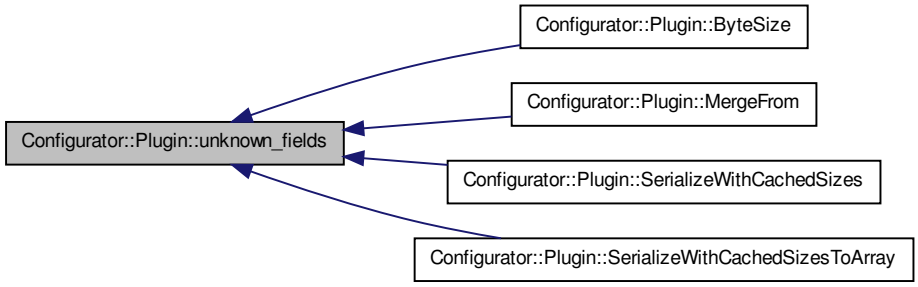
void Configurator::Plugin::set_weight ([double]value) [inline]

Here is the caller graph for this function:

void Configurator::Plugin::Swap ([Plugin ∗]other)

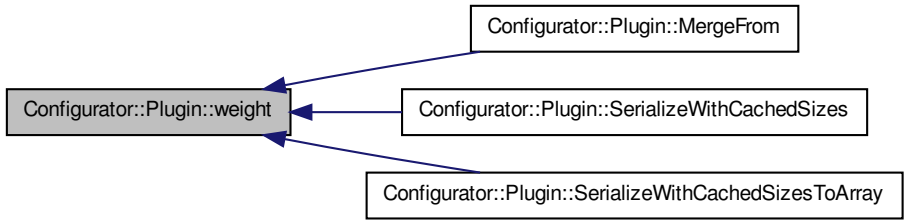
const ::google::protobuf::UnknownFieldSet& Configurator::Plugin::unknown_fields ( ) const [inline]

Here is the caller graph for this function:



double Configurator::Plugin::weight ( ) const [inline]

Here is the caller graph for this function:



7.55.7 Friends And Related Function Documentation

void protobuf_AddDesc_confloadbalancer_2eproto ( ) [friend]

void protobuf_AssignDesc_confloadbalancer_2eproto ( ) [friend]

void protobuf_ShutdownFile_confloadbalancer_2eproto ( ) [friend]

7.55.8 Member Data Documentation

const int Configurator::Plugin::kNameFieldNumber = 1 [static]

const int Configurator::Plugin::kWeightFieldNumber = 2 [static]

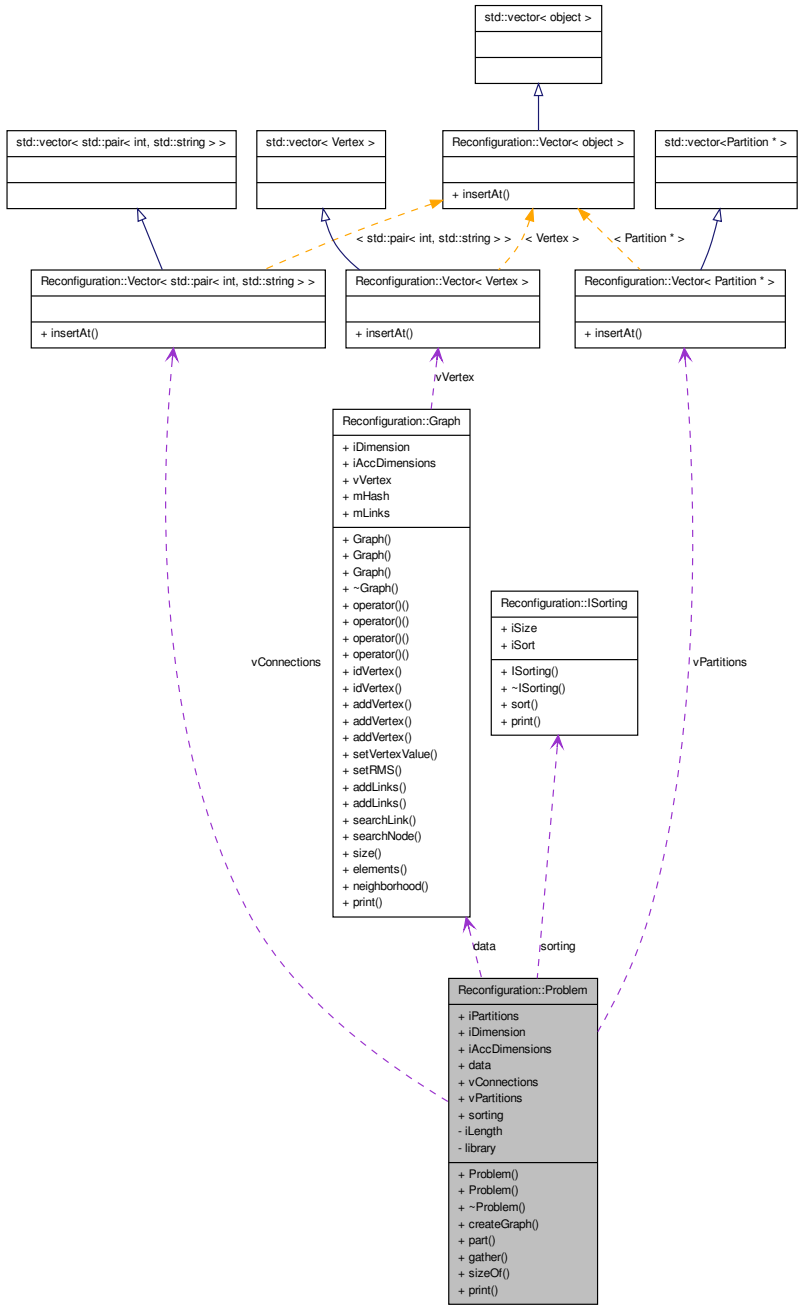
The documentation for this class was generated from the following files:

- confloadbalancer.pb.h
- confloadbalancer.pb.cc

7.56 Reconfiguration::Problem Class Reference

`#include <Problem.h>`

Collaboration diagram for Reconfiguration::Problem:



7.56.1 Public Member Functions

- Problem ()

  Constructor of the Problem class.

- Problem (int, int, int ∗, std::map< std::string, Coordinate >, std::map< std::string, int >, Vector< std::pair< int, std::string > >, bool ∗, std::string, std::string, Algorithm ∗)

  Constructor of the Problem class.

- virtual ∼Problem ()

  virtual destructor of the class Problem.

- void createGraph (bool ∗, std::map< std::string, Coordinate >, int)

  Creates and configures the application graph.

- void part (Vector< int >)

  Parts the problem in several domains (partitions).

- void gather ()

  Gathers the results of all slaves.

- int sizeOf ()

  Calculates the size (in Bytes) of the application graph.

- void print ()

  prints the Problem object.

7.56.2 Public Attributes

- int iPartitions
- int iDimension
- int ∗ iAccDimensions
- Graph ∗ data
- Vector< std::pair< int, std::string > > vConnections
- Vector< Partition ∗ > vPartitions

- ISorting * sorting

7.56.3 Detailed Description

Allocates information about the problem that must be solved by the slaves from the point of view of the master process. It contains the application graph, the partitions vector and the sorting plugins for the domain decomposition in spite of information to create the application graph.

7.56.4 Constructor & Destructor Documentation

Reconfiguration::Problem::Problem ( )

Constructor of the Problem class.

Returns

*this

Creates an object Problem. Initiates all pointers to NULL.

Reconfiguration::Problem::Problem ([int]iPartitions, int *iDimension,* int * *iAccDimensions,* std::map< std::string, Coordinate > *mCells,* std::map< std::string, int > *mLinks,* Vector< std::pair< int, std::string > > *vConnections,* bool * *bBoundaries,* std::string *sName,* std::string *sSort,* Algorithm * *algorithm*)

Constructor of the Problem class.

Parameters

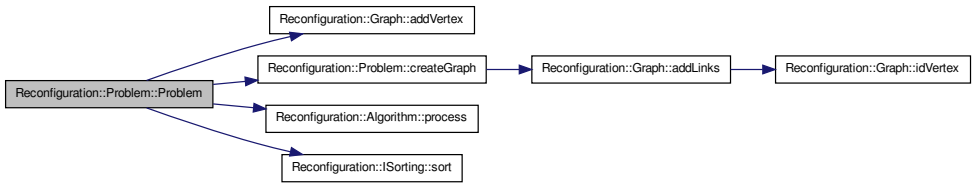
in	*iPartitions*	Number of partitions to be made on the graph.
in	*iDimension*	Problem dimension.
in	*iAccDimensions*	Array of accumulated length per problem dimension. The last position of this array must be equal to the number of vertices of the graph.
in	*mCells*	Pairs between link name and set of coordenates for that link.
in	*mLinks*	Pairs between links names and global offset for that link. This offset depends on the graph dimensions.
in	*vConnections*	Pairs between vertex identifiers and link names.
in	*bBoundaries*	Boundary policy per problems dimension.
in	*sName*	Data input values filename for configuring the application graph of the problem.
in	*sSort*	Sort method name.
in	*algorithm*	Algorithm object to be solved.

Returns

∗this

Creates an object Problem intializing the application graph (reading its values from an input file) and allocating memory for the partition vector. Creates also the vertex arrangement calling the user specified plugin.

Here is the call graph for this function:



Reconfiguration::Problem::∼Problem ( ) `[virtual]`

virtual destructor of the class Problem.

Returns

void

Destroys the Problem object. Frees the dynamic library.

7.56.5 Member Function Documentation

void Reconfiguration::Problem::createGraph ([bool ∗]bBoundaries, std::map< std::string, Coordinate > *mCells,* int *iDimension*)

Creates and configures the application graph.

Parameters

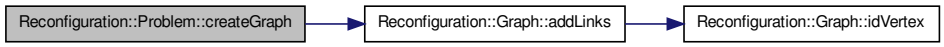
in	*bBoundaries*	Boundary policy per problems dimension.
in	*mCells*	Pairs between link name and set of coordenates for that link. This offset depends on the graph dimensions.
in	*iDimension*	Problem dimension.

Returns

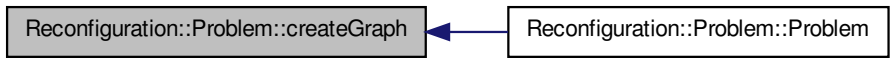
void

Creates and configures the application graph.

Here is the call graph for this function:



Here is the caller graph for this function:



void Reconfiguration::Problem::gather ( )

Gathers the results of all slaves.

Returns

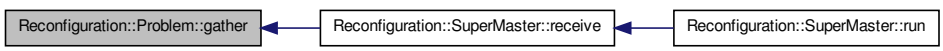
void

Gathers the results obtained by the slaves taking into account only the updated values while the step execution (RW values).

See also

UPDATE

Here is the caller graph for this function:



void Reconfiguration::Problem::part ([Vector< int >]iRMS)

Parts the problem in several domains (partitions).

Parameters

in	*iRMS*	RMS value of each slave (length of RW values).

Returns

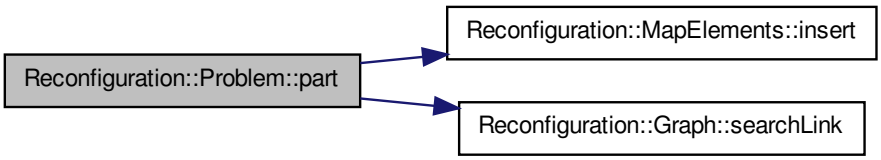
void

Parts a problem in several independent domains knowing the number of RW tasks that each slave must receive.

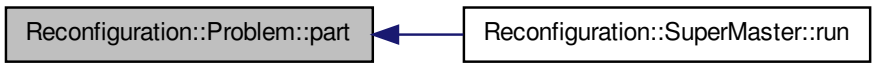
See also

UPDATE

Here is the call graph for this function:



Here is the caller graph for this function:



void Reconfiguration::Problem::print ( )

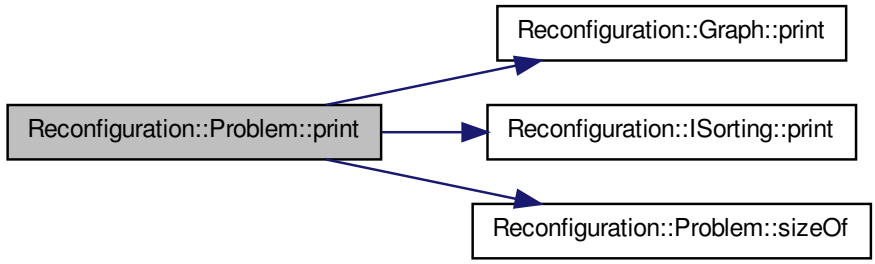
prints the Problem object.

Returns

void

Prints the Problem object.

Here is the call graph for this function:



int Reconfiguration::Problem::sizeOf ( )

Calculates the size (in Bytes) of the application graph.

Returns

int

Calculates the size (in Bytes) of the application graph.

Here is the caller graph for this function:

7.56.6 Member Data Documentation

Graph * Reconfiguration::Problem::data

Application graph.

int * Reconfiguration::Problem::iAccDimensions

iAccDimensions Array of accumulated length per problem dimension. The last position of this array must be equal to the number of vertices of the graph.

int Reconfiguration::Problem::iDimension

Problem dimension.

int Reconfiguration::Problem::iPartitions

Number of partitions to be made on the graph.

ISorting * Reconfiguration::Problem::sorting

Decomposition plugin specified by the user.

Vector< std::pair< int, std::string > > Reconfiguration::Problem::vConnections

Pairs between vertex identifiers and link names.

Vector< Partition * > Reconfiguration::Problem::vPartitions

Partitions set of the graph.

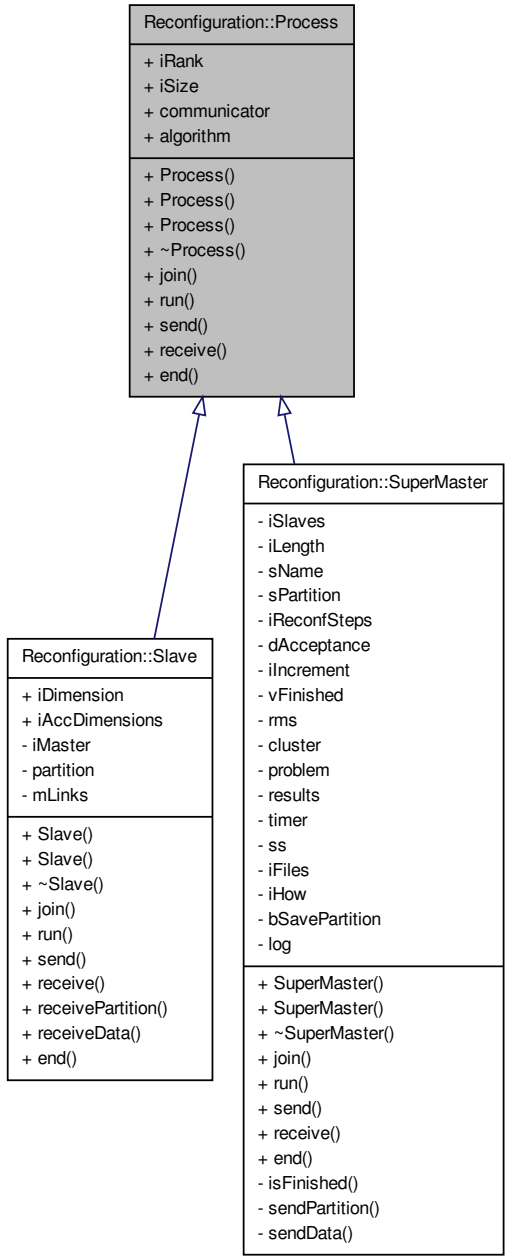
The documentation for this class was generated from the following files:

- Problem.h
- Problem.cpp

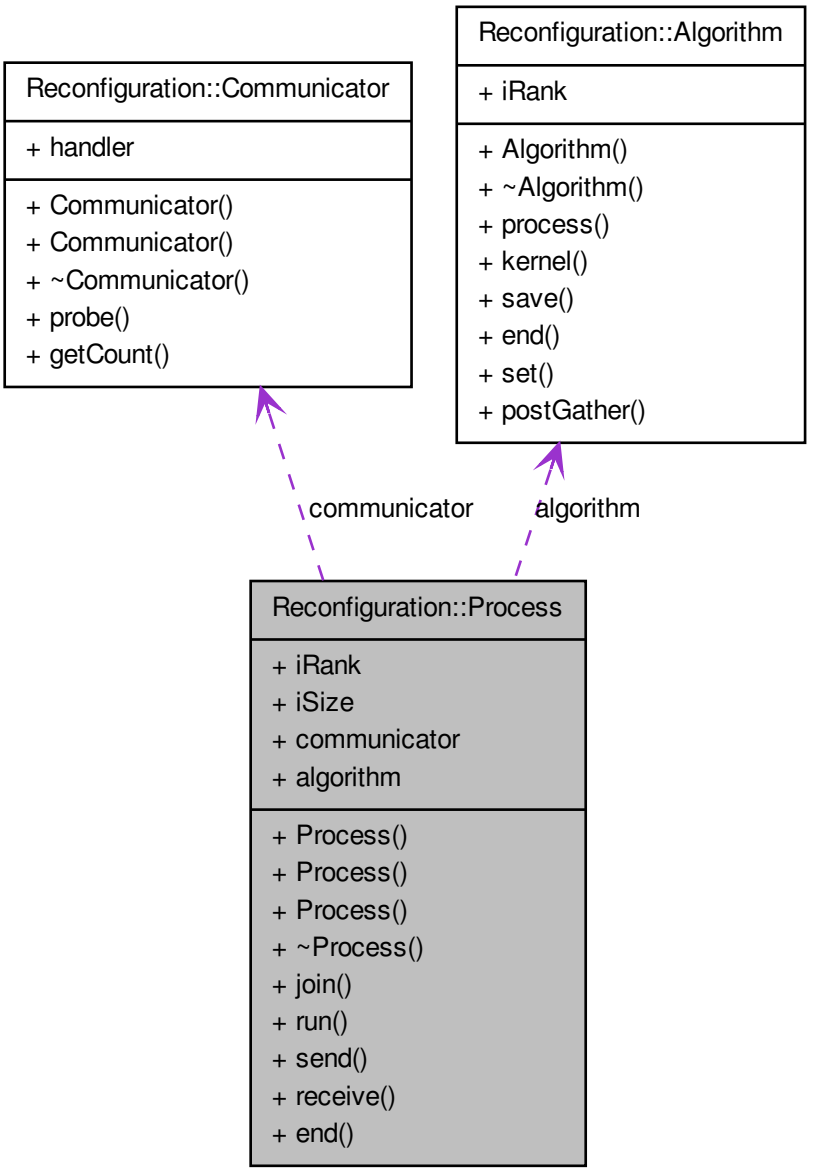
7.57 Reconfiguration::Process Class Reference

`#include <Process.h>`

Inheritance diagram for Reconfiguration::Process:



Collaboration diagram for Reconfiguration::Process:



7.57.1 Public Member Functions

- Process ()

  Constructor of the Process class.

- Process (int, Algorithm ∗)

  Constructor of the Process class.

- Process (int, Algorithm ∗, int)

  Constructor of the Process class.

- virtual ∼Process ()

  virtual destructor of the class Process.

- virtual void join ()=0

  Virtual join method.

- virtual void run ()=0

  Virtual run method.

- virtual void send (int)=0

  Virtual send method.

- virtual void receive ()=0

  Virtual receive method.

- virtual void end ()=0

  Virtual end method.

7.57.2 Public Attributes

- int iRank
- int iSize
- Communicator ∗ communicator
- Algorithm ∗ algorithm

7.57.3 Detailed Description

Abstract class to manage the processes.

7.57.4 Constructor & Destructor Documentation

Reconfiguration::Process::Process ( )

Constructor of the Process class.

Returns

∗this

Creates an object Process. Sets the pointers to NULL.

Reconfiguration::Process::Process ([int]iRank, Algorithm ∗ *algorithm*)

Constructor of the Process class.

Parameters

in	*iRank*	MPI Rank of the process.
in	*algorithm*	Pointer to the algorithm object to be solved.

Returns

∗this

Creates an object Process. Sets the MPI rank, the MPI size, the algorithm and the communicator channel.

Reconfiguration::Process::Process ([int]iRank, Algorithm ∗ *algorithm,* int *iMaster*)

Constructor of the Process class.

Parameters

in	*iRank*	MPI Rank of the process.
in	*algorithm*	Pointer to the algorithm object to be solved.
in	*iMaster*	Father process rank.

Returns

∗this

Creates an object Process. Sets the MPI rank, the algorithm and the communicator channel.

Todo Create a communicator channel with my master process.

Parameters

in	*iRank*	MPI Rank of the process.
in	*algorithm*	Pointer to the algorithm object to be solved.
in	*iMaster*	Father process rank.

Returns

∗this

Creates an object Process. Sets the MPI rank, the algorithm and the communicator channel.

Todo Create a communicator channel with my master process.

Reconfiguration::Process::∼Process ( ) [virtual]

virtual destructor of the class Process.

Returns

void

Destroys the Process object.

7.57.5 Member Function Documentation

virtual void Reconfiguration::Process::end ( ) [pure virtual]

Virtual end method.

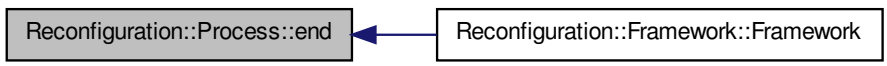
Returns

void

Implemented by the SuperMaster or Slave process.

Implemented in Reconfiguration::SuperMaster, and Reconfiguration::Slave.

Here is the caller graph for this function:



virtual void Reconfiguration::Process::join ( ) [pure virtual]

Virtual join method.

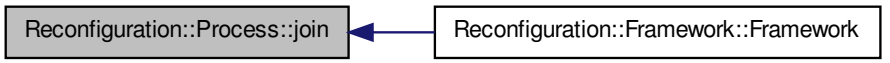
Returns

void

Implemented by the SuperMaster or Slave process.

Implemented in Reconfiguration::SuperMaster, and Reconfiguration::Slave.

Here is the caller graph for this function:



virtual void Reconfiguration::Process::receive ( ) [pure virtual]

Virtual receive method.

Returns

void

Implemented by the SuperMaster or Slave process.

Implemented in Reconfiguration::SuperMaster, and Reconfiguration::Slave.

virtual void Reconfiguration::Process::run ( ) [pure virtual]

Virtual run method.

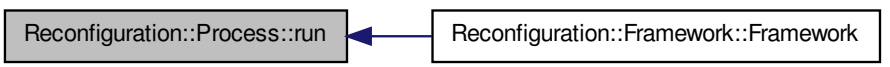
Returns

void

Implemented by the SuperMaster or Slave process.

Implemented in Reconfiguration::SuperMaster, and Reconfiguration::Slave.

Here is the caller graph for this function:



virtual void Reconfiguration::Process::send ([int]) [pure virtual]

Virtual send method.

Returns

void

Implemented by the SuperMaster or Slave process.

Implemented in Reconfiguration::SuperMaster, and Reconfiguration::Slave.

7.57.6 Member Data Documentation

Algorithm * Reconfiguration::Process::algorithm

Pointer to the algorithm object to be solved.

Communicator * Reconfiguration::Process::communicator

Communication handler between processes.

int Reconfiguration::Process::iRank

Process rank.

int Reconfiguration::Process::iSize

Number of copies launched.

The documentation for this class was generated from the following files:

- Process.h
- Process.cpp

7.58 Reconfiguration::Queue Class Reference

`#include <IGrouping.h>`

7.58.1 Public Member Functions

- Queue ()

  Constructor of the Queue class.

- virtual ∼Queue ()

  virtual destructor of the class Queue.

- int size ()

  Calculates the number of elements in a queue.

- void print ()

  prints the Queue object.

- void addItem (int, int, double, double)

  Adds a new element to the buffer queue.

- void addItem (Item)

  Adds a new element to the visited queue.

- Item extractMin ()

  Extracts the element with the minimum weight.

7.58.2 Detailed Description

OpenList and CloseList to implement the grouping algorithms.

7.58.3 Constructor & Destructor Documentation

Reconfiguration::Queue::Queue ( )

Constructor of the Queue class.

Returns

∗this

Creates an object Queue.

Reconfiguration::Queue::∼Queue ( ) `[virtual]`

virtual destructor of the class Queue.

Returns

void

Destroys the Queue object.

7.58.4 Member Function Documentation

void Reconfiguration::Queue::addItem ([int]iId, int *iIdOr,* double *dWeight,* double *dAccWeight*)

Adds a new element to the buffer queue.

Parameters

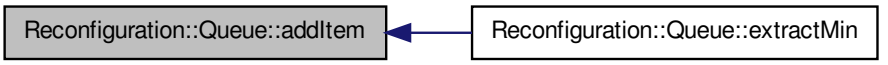
in	*iId*	Vertex identifier.
in	*iIdOr*	Source Vertex identifier.
in	*dWeight*	Vertex weight.
in	*dAccWeight*	Accumulated weight from the beginning of the path.

Returns

void

Adds a new Item element to the buffer queue creating it before.

Here is the caller graph for this function:



void Reconfiguration::Queue::addItem ([Item]itemAux)

Adds a new element to the visited queue.

Parameters

in	*itemAux*	object to add.

Returns

void

Adds a new Item element to the visited queue.

Item Reconfiguration::Queue::extractMin ( )

Extracts the element with the minimum weight.

Returns

Item

Extracts a copy of the element with the minimum weight.

Here is the call graph for this function:

void Reconfiguration::Queue::print ( )

prints the Queue object.

Returns

void

Prints the Queue object printing its elements.

int Reconfiguration::Queue::size ( )

Calculates the number of elements in a queue.

Returns

int

Calculates the size of the buffer queue.

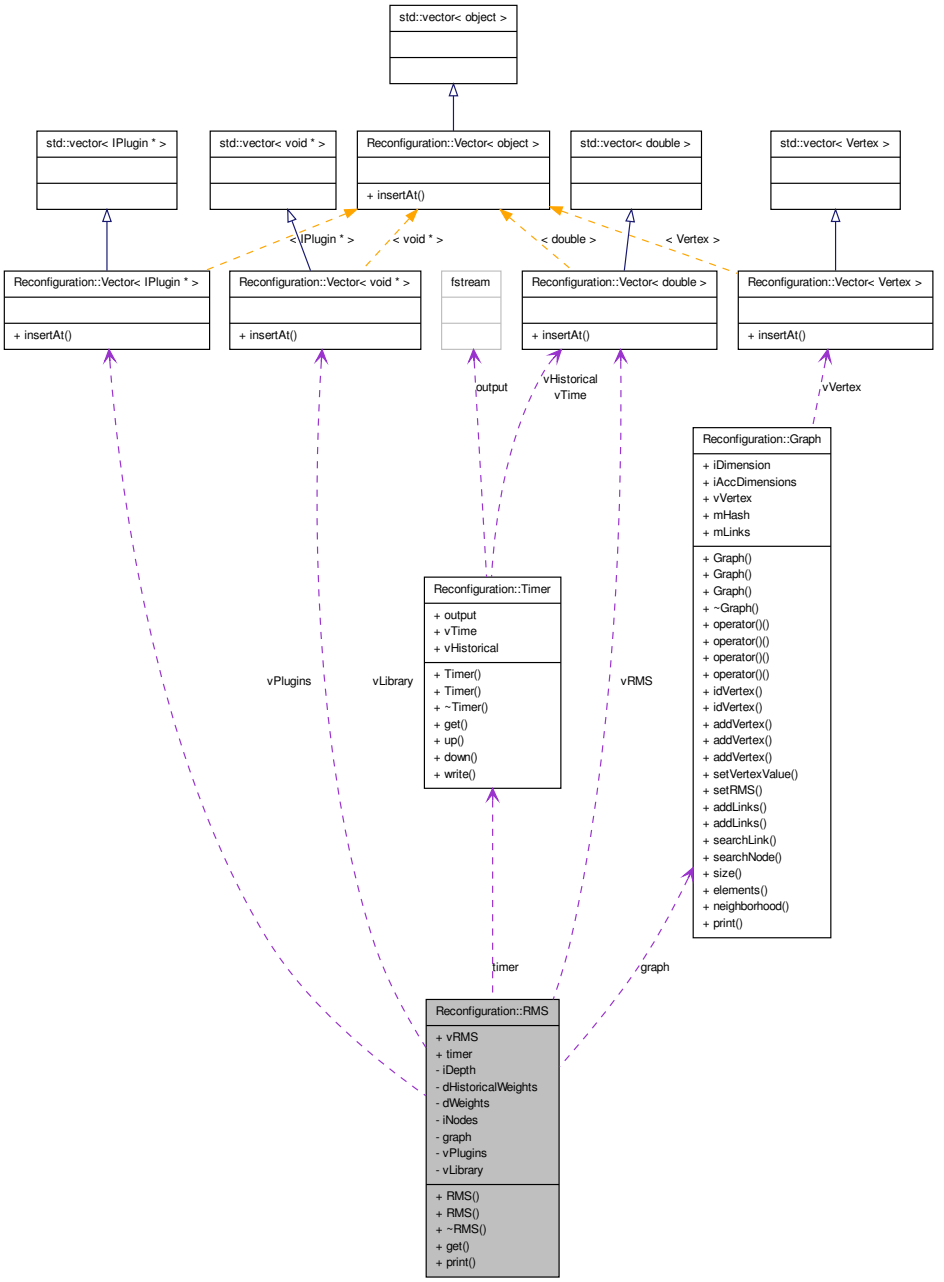
The documentation for this class was generated from the following files:

- IGrouping.h
- IGrouping.cpp

7.59 Reconfiguration::RMS Class Reference

`#include <RMS.h>`

Collaboration diagram for Reconfiguration::RMS:



7.59.1 Public Member Functions

- RMS ()

  Constructor of the RMS class.

- RMS (Graph ∗, int, int, std::string ∗, double ∗, double ∗, std::string)

  Constructor of the RMS class.

- virtual ∼RMS ()

  virtual destructor of the class RMS.

- void get (int)

  Calls the get method of each loadbalancing plugin.

- void print ()

  prints the RMS object.

7.59.2 Public Attributes

- Vector< double > vRMS
- Timer ∗ timer

7.59.3 Detailed Description

Manages the loadbalancing plugins.

7.59.4 Constructor & Destructor Documentation

Reconfiguration::RMS::RMS ( )

Constructor of the RMS class.

Returns

∗this

Creates an object RMS.

Reconfiguration::RMS::RMS ([Graph ∗]graph, int *iPlugins,* int *iDepth,* std::string ∗ *sPluginNames,* double ∗ *dWeights,* double ∗ *dHistoricalWeights,* std::string *sTime*)

Constructor of the RMS class.

Parameters

in	*graph*	System graph.
in	*iPlugins*	Number of plugins specified by the user.
in	*iDepth*	Number of historical values to take into account to calculate the loadbalancing policies, if any time plugin is involved.
in	*sPluginNames*	Names of the plugins.
in	*dWeights*	Contribution of each plugin to the global RMS value.
in	*dHistoricalWeights*	Weight of each historical value from the point of view of the global weight for the temporal plugin. Related to the values read from the user configuration file.
in	*sTime*	Time log filename.

Returns

∗this

Creates an object RMS.

Here is the call graph for this function:

Reconfiguration::RMS::∼RMS ( ) [virtual]

virtual destructor of the class RMS.

Returns

void

Destroys the RMs object.

Returns

void

Destroys the RMS object.

7.59.5 Member Function Documentation

void Reconfiguration::RMS::get ([int]iLength)

Calls the get method of each loadbalancing plugin.

Parameters

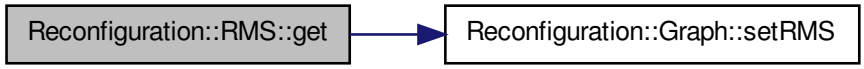
in	*iLength*	Length of the problem to normalize the RMS value to the global number of tasks of the application graph.

Returns

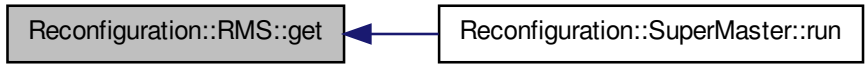
void

Invokes the get method of each loaded plugin and takes their results to calculate the final RMS value for this node.

Here is the call graph for this function:



Here is the caller graph for this function:



void Reconfiguration::RMS::print ( )

prints the RMS object.

Returns

void

Prints the RMS object.

7.59.6 Member Data Documentation

Timer * Reconfiguration::RMS::timer

Takes the execution time of each slave.

Vector< double > Reconfiguration::RMS::vRMS

Vector of RMS values (how many tasks must execute each slave node).

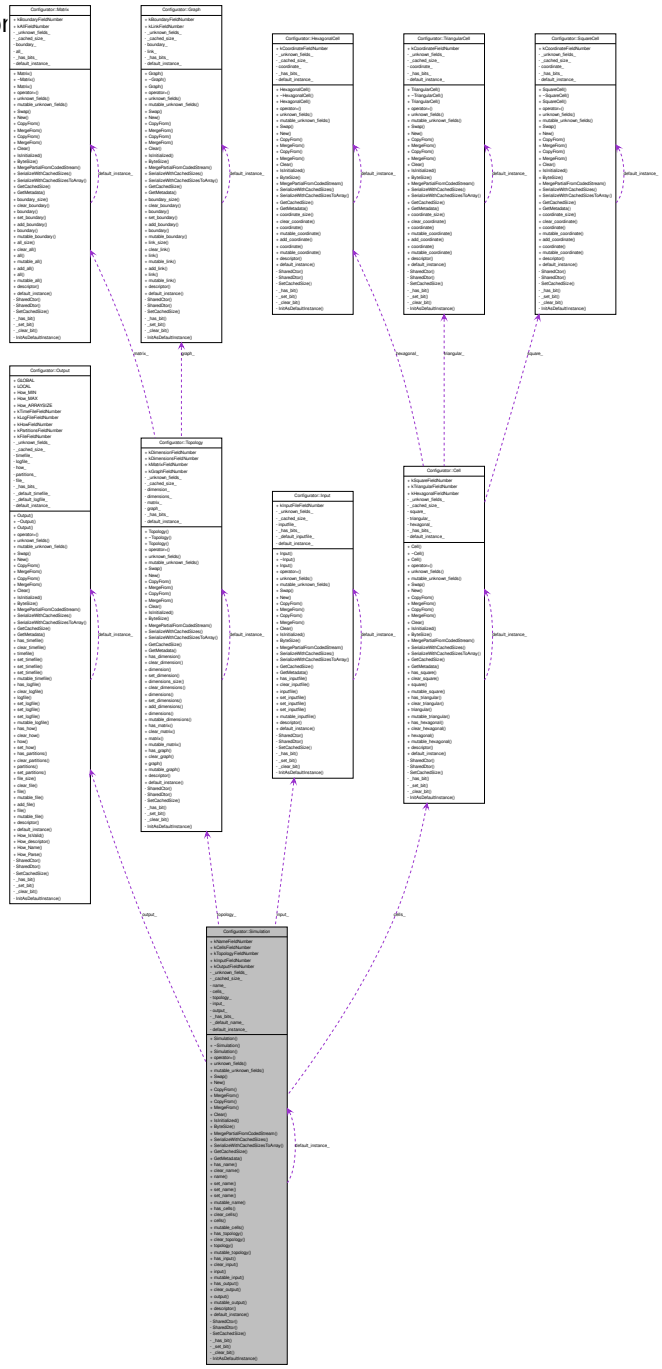
The documentation for this class was generated from the following files:

- RMS.h
- RMS.cpp

7.60 Configurator::Simulation Class Reference

`#include <confproblem.pb.h>`

Collaboration



7.60.1 Public Member Functions

- Simulation ()
- virtual ∼Simulation ()
- Simulation (const Simulation &from)
- Simulation & operator= (const Simulation &from)
- const ::google::protobuf::UnknownFieldSet & unknown_fields () const
- inline::google::protobuf::UnknownFieldSet ∗ mutable_unknown_fields ()
- void Swap (Simulation ∗other)
- Simulation ∗ New () const
- void CopyFrom (const ::google::protobuf::Message &from)
- void MergeFrom (const ::google::protobuf::Message &from)
- void CopyFrom (const Simulation &from)
- void MergeFrom (const Simulation &from)
- void Clear ()
- bool IsInitialized () const
- int ByteSize () const
- bool MergePartialFromCodedStream (::google::protobuf::io::CodedInputStream ∗input)
- void SerializeWithCachedSizes (::google::protobuf::io::CodedOutputStream ∗output) const
- ::google::protobuf::uint8 ∗ SerializeWithCachedSizesToArray (::google::protobuf::uint8 ∗output) const
- int GetCachedSize () const
- ::google::protobuf::Metadata GetMetadata () const
- bool has_name () const
- void clear_name ()
- const ::std::string & name () const
- void set_name (const ::std::string &value)
- void set_name (const char ∗value)
- void set_name (const char ∗value, size_t size)
- inline::std::string ∗ mutable_name ()
- bool has_cells () const
- void clear_cells ()
- const ::Configurator::Cell & cells () const

- inline::Configurator::Cell * mutable_cells ()
- bool has_topology () const
- void clear_topology ()
- const ::Configurator::Topology & topology () const
- inline::Configurator::Topology * mutable_topology ()
- bool has_input () const
- void clear_input ()
- const ::Configurator::Input & input () const
- inline::Configurator::Input * mutable_input ()
- bool has_output () const
- void clear_output ()
- const ::Configurator::Output & output () const
- inline::Configurator::Output * mutable_output ()

7.60.2 Static Public Member Functions

- static const ::google::protobuf::Descriptor * descriptor ()
- static const Simulation & default_instance ()

7.60.3 Static Public Attributes

- static const int kNameFieldNumber = 1
- static const int kCellsFieldNumber = 2
- static const int kTopologyFieldNumber = 3
- static const int kInputFieldNumber = 4
- static const int kOutputFieldNumber = 5

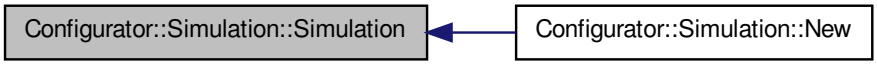
7.60.4 Friends

- void protobuf_AddDesc_confproblem_2eproto ()
- void protobuf_AssignDesc_confproblem_2eproto ()
- void protobuf_ShutdownFile_confproblem_2eproto ()

7.60.5 Constructor & Destructor Documentation

Configurator::Simulation::Simulation ( )

Here is the caller graph for this function:



Configurator::Simulation::~Simulation ( ) [virtual]

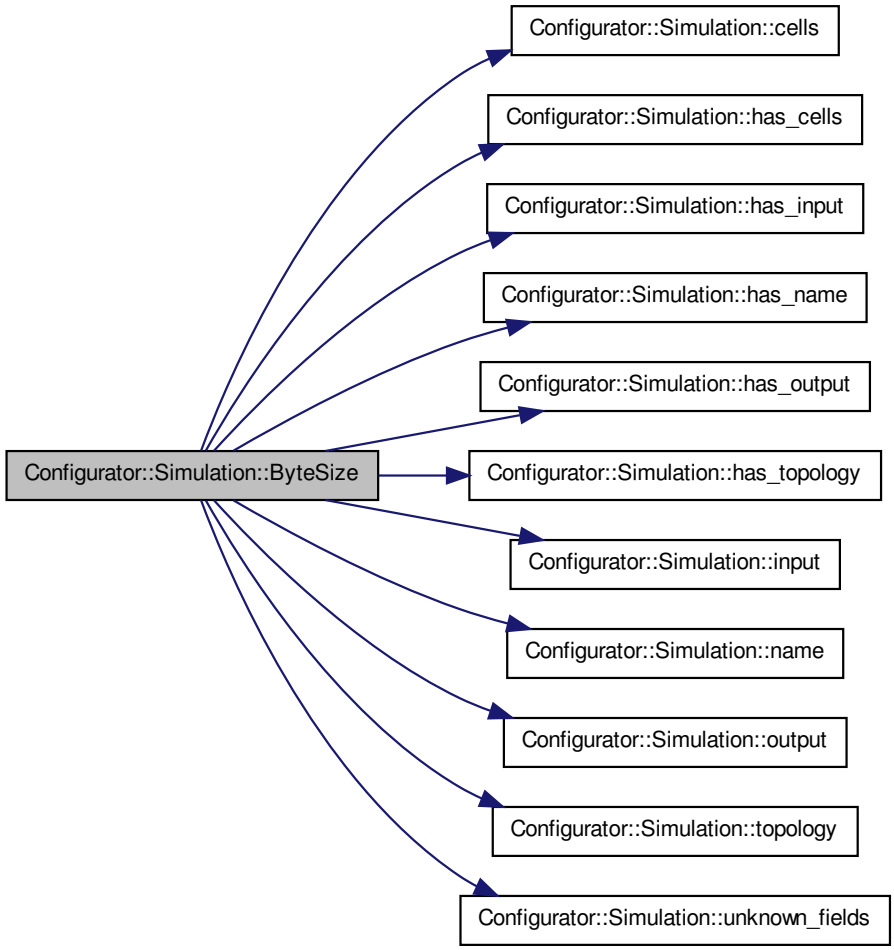
Configurator::Simulation::Simulation ([const Simulation &]from)

Here is the call graph for this function:

7.60.6 Member Function Documentation

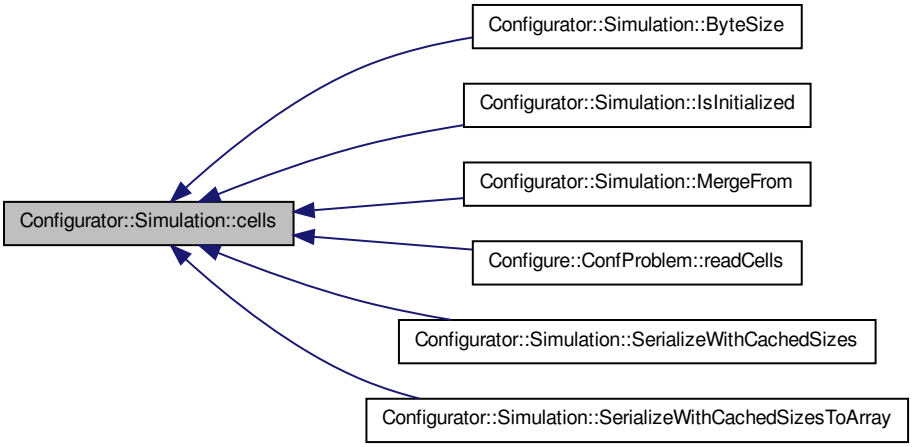
int Configurator::Simulation::ByteSize ( ) const

Here is the call graph for this function:



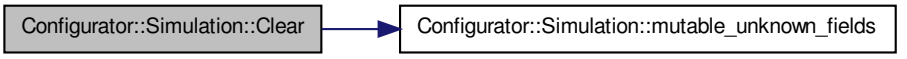
const ::Configurator::Cell & Configurator::Simulation::cells ( ) const `[inline]`

Here is the caller graph for this function:

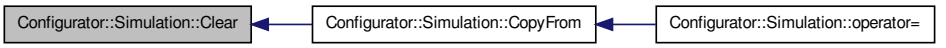


void Configurator::Simulation::Clear ( )

Here is the call graph for this function:



Here is the caller graph for this function:



void Configurator::Simulation::clear_cells ( ) [inline]

void Configurator::Simulation::clear_input ( ) [inline]

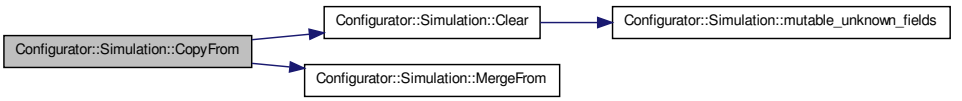
void Configurator::Simulation::clear_name ( ) [inline]

void Configurator::Simulation::clear_output ( ) [inline]

void Configurator::Simulation::clear_topology ( ) [inline]

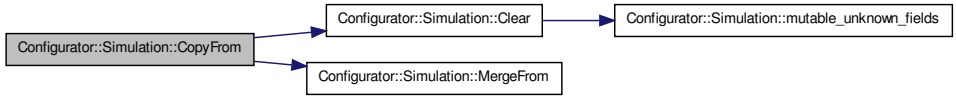
void Configurator::Simulation::CopyFrom ([const Simulation &]from)

Here is the call graph for this function:



void Configurator::Simulation::CopyFrom ([const ::google::protobuf::Message &]from)

Here is the call graph for this function:



Here is the caller graph for this function:

const Simulation & Configurator::Simulation::default_instance ( ) [static]

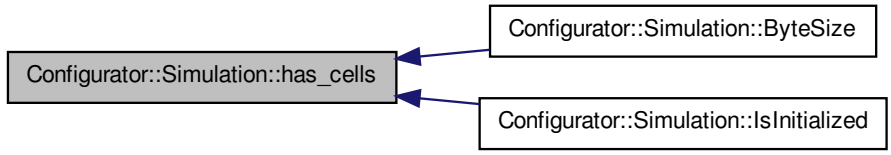
const ::google::protobuf::Descriptor * Configurator::Simulation::descriptor ( ) [static]

int Configurator::Simulation::GetCachedSize ( ) const [inline]

google::protobuf::Metadata Configurator::Simulation::GetMetadata ( ) const

bool Configurator::Simulation::has_cells ( ) const [inline]

Here is the caller graph for this function:



bool Configurator::Simulation::has_input ( ) const [inline]

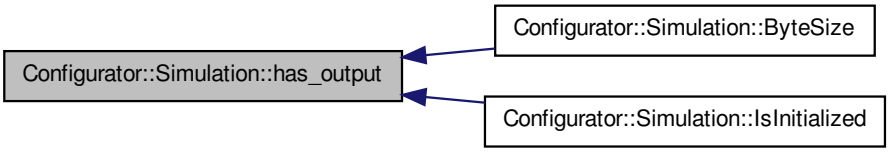
Here is the caller graph for this function:

bool Configurator::Simulation::has_name ( ) const [inline]

Here is the caller graph for this function:

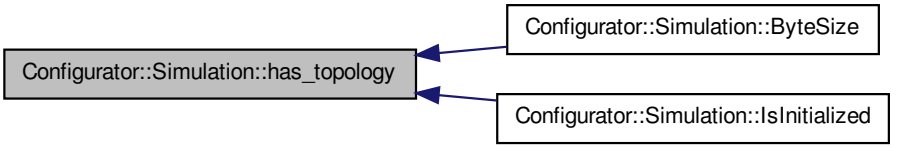
bool Configurator::Simulation::has_output ( ) const [inline]

Here is the caller graph for this function:



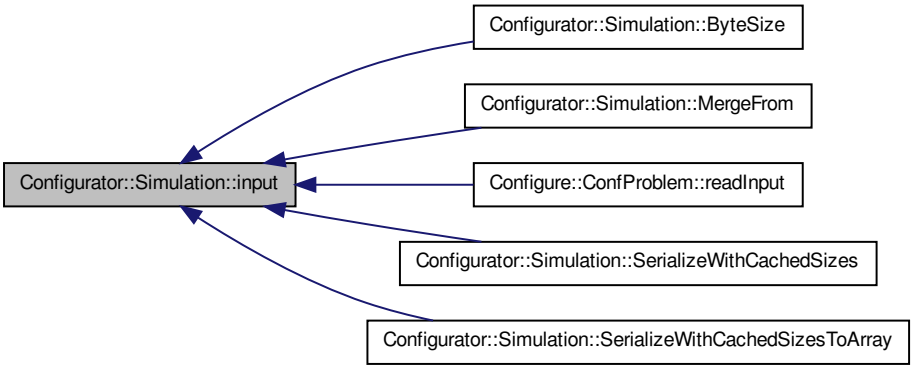
bool Configurator::Simulation::has_topology ( ) const [inline]

Here is the caller graph for this function:



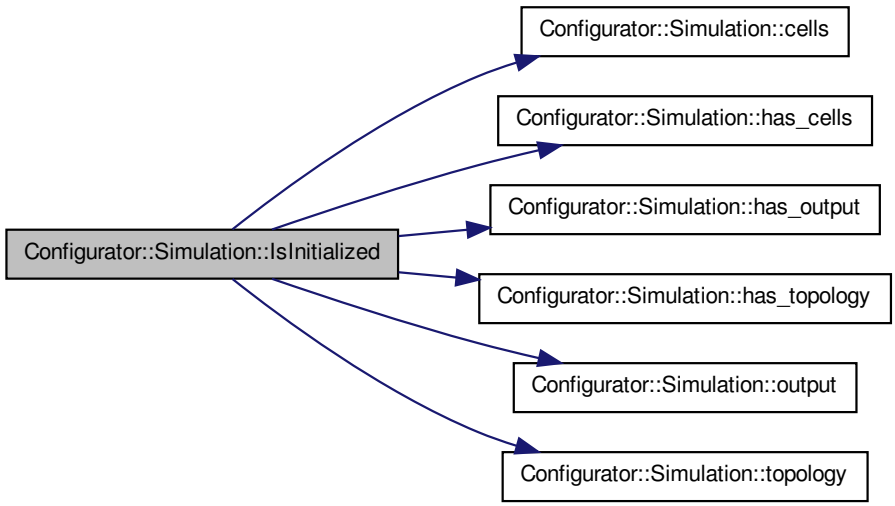
const ::Configurator::Input & Configurator::Simulation::input ( ) const [inline]

Here is the caller graph for this function:



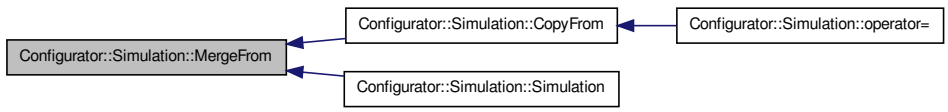
bool Configurator::Simulation::IsInitialized ( ) const

Here is the call graph for this function:



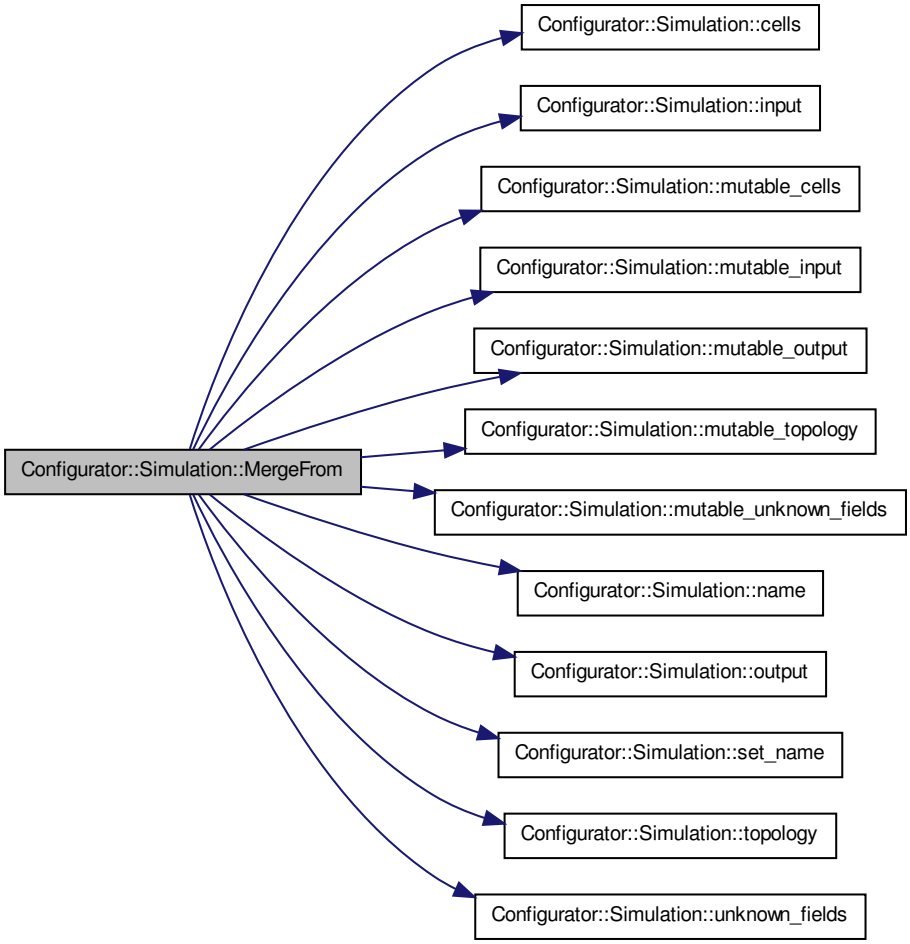
void Configurator::Simulation::MergeFrom ([const ::google::protobuf::Message &]from)

Here is the caller graph for this function:



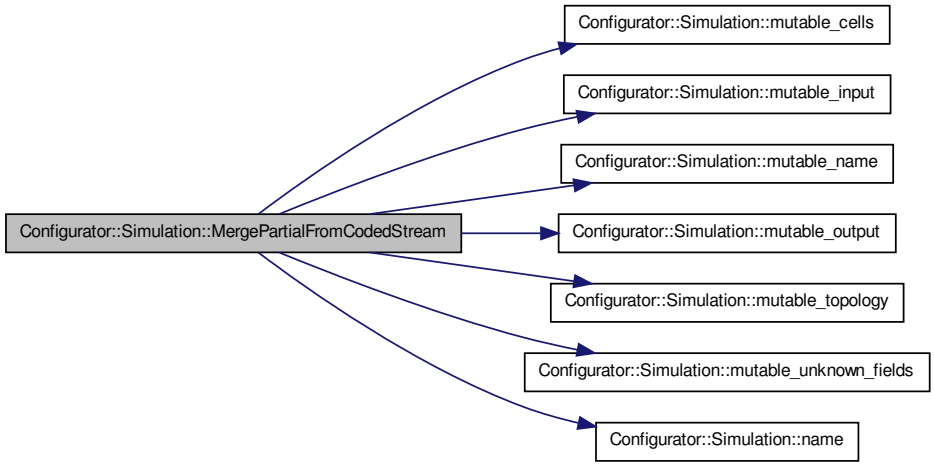
void Configurator::Simulation::MergeFrom ([const Simulation &]from)

Here is the call graph for this function:



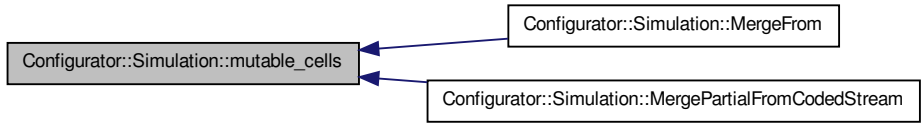
bool Configurator::Simulation::MergePartialFromCodedStream ([::google::protobuf::io::CodedInputStream *]input)

Here is the call graph for this function:



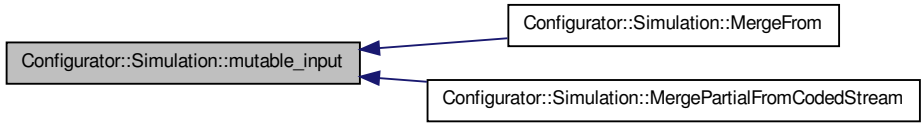
Configurator::Cell * Configurator::Simulation::mutable_cells ( ) [inline]

Here is the caller graph for this function:



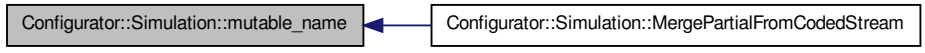
Configurator::Input * Configurator::Simulation::mutable_input ( ) [inline]

Here is the caller graph for this function:



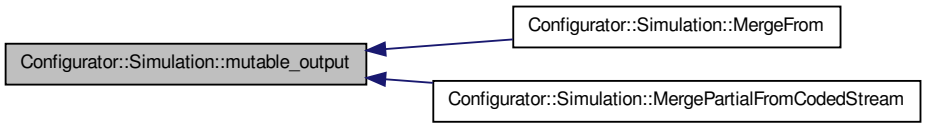
std::string * Configurator::Simulation::mutable_name ( ) [inline]

Here is the caller graph for this function:



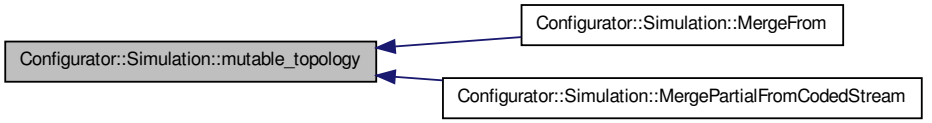
Configurator::Output * Configurator::Simulation::mutable_output ( ) [inline]

Here is the caller graph for this function:



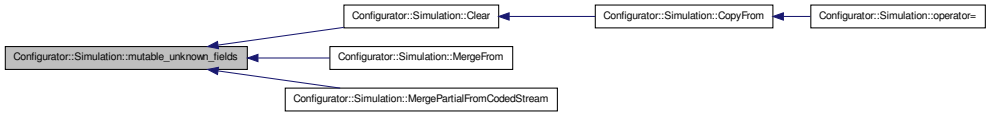
Configurator::Topology * Configurator::Simulation::mutable_topology ( ) [inline]

Here is the caller graph for this function:



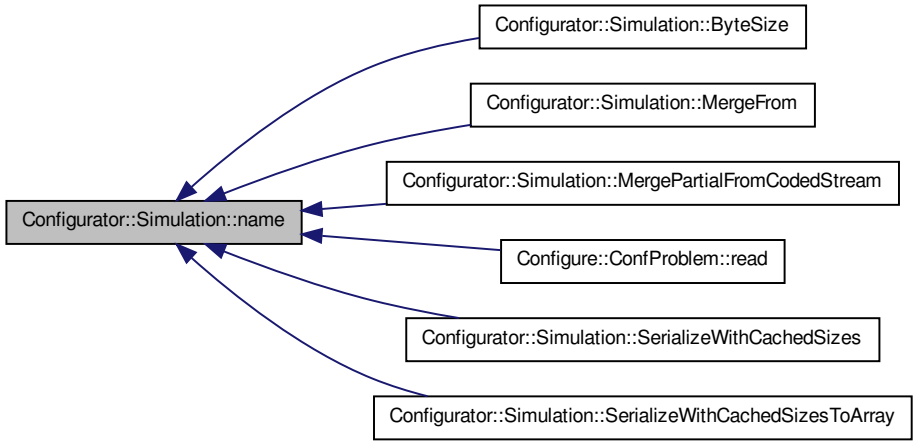
inline ::google::protobuf::UnknownFieldSet* Configurator::Simulation::mutable_-unknown_fields ( ) [inline]

Here is the caller graph for this function:



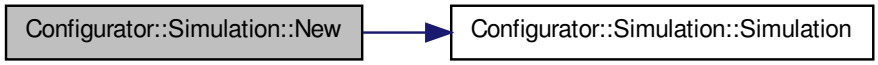
const ::std::string & Configurator::Simulation::name ( ) const [inline]

Here is the caller graph for this function:



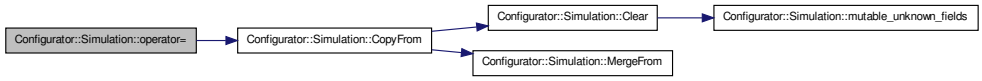
Simulation * Configurator::Simulation::New ( ) const

Here is the call graph for this function:



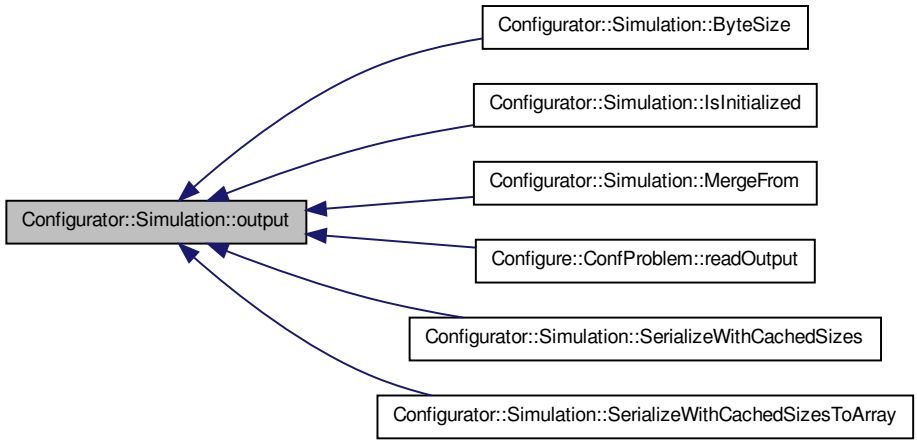
Simulation& Configurator::Simulation::operator= ([const Simulation &]from) `[inline]`

Here is the call graph for this function:



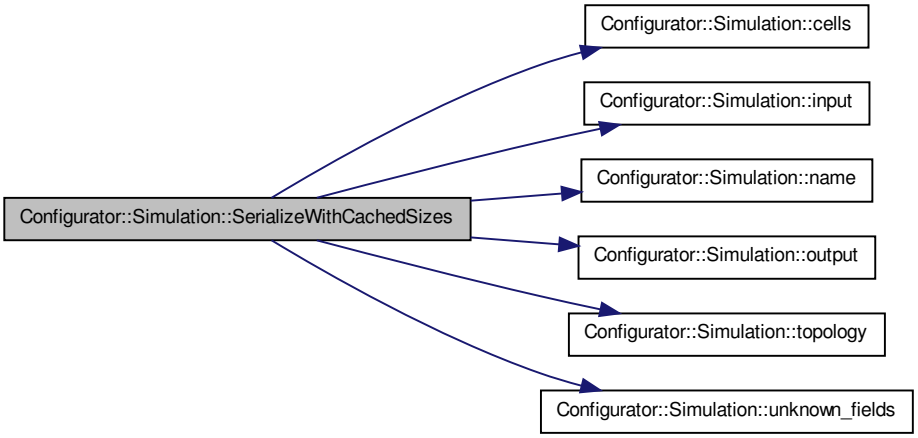
const ::Configurator::Output & Configurator::Simulation::output ( ) const `[inline]`

Here is the caller graph for this function:



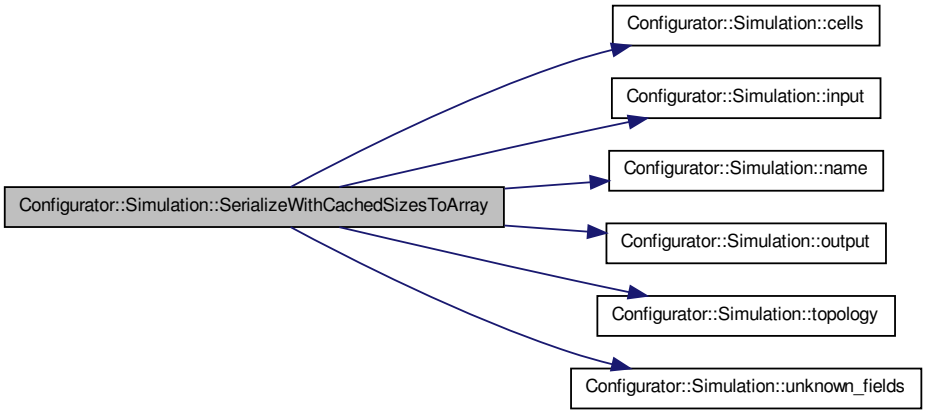
void Configurator::Simulation::SerializeWithCachedSizes ([::google::protobuf::io::CodedOutputStream *]output) const

Here is the call graph for this function:



google::protobuf::uint8 * Configurator::Simulation::SerializeWithCachedSizesToArray ([::google::protobuf::uint8 *]output) const

Here is the call graph for this function:



void Configurator::Simulation::set_name ([const char *]value, size_t *size*) `[inline]`

void Configurator::Simulation::set_name ([const ::std::string &]value) `[inline]`

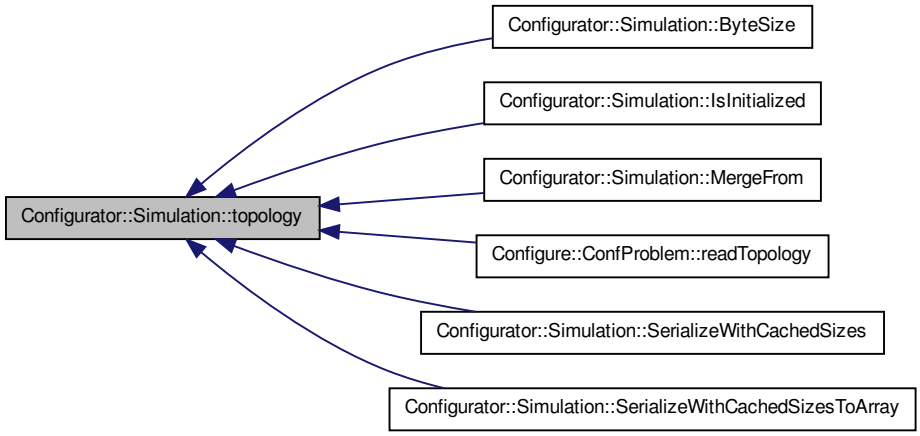
Here is the caller graph for this function:

void Configurator::Simulation::set_name ([const char ∗]value) [inline]

void Configurator::Simulation::Swap ([Simulation ∗]other)

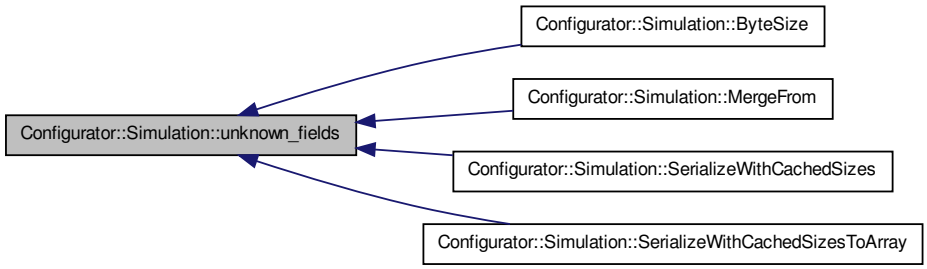
const ::Configurator::Topology & Configurator::Simulation::topology ( ) const [inline]

Here is the caller graph for this function:



const ::google::protobuf::UnknownFieldSet& Configurator::Simulation::unknown_fields ( ) const [inline]

Here is the caller graph for this function:



7.60.7 Friends And Related Function Documentation

void protobuf_AddDesc_confproblem_2eproto ( ) [friend]

void protobuf_AssignDesc_confproblem_2eproto ( ) [friend]

void protobuf_ShutdownFile_confproblem_2eproto ( ) [friend]

7.60.8 Member Data Documentation

const int Configurator::Simulation::kCellsFieldNumber = 2 `[static]`

const int Configurator::Simulation::kInputFieldNumber = 4 `[static]`

const int Configurator::Simulation::kNameFieldNumber = 1 `[static]`

const int Configurator::Simulation::kOutputFieldNumber = 5 `[static]`

const int Configurator::Simulation::kTopologyFieldNumber = 3 `[static]`

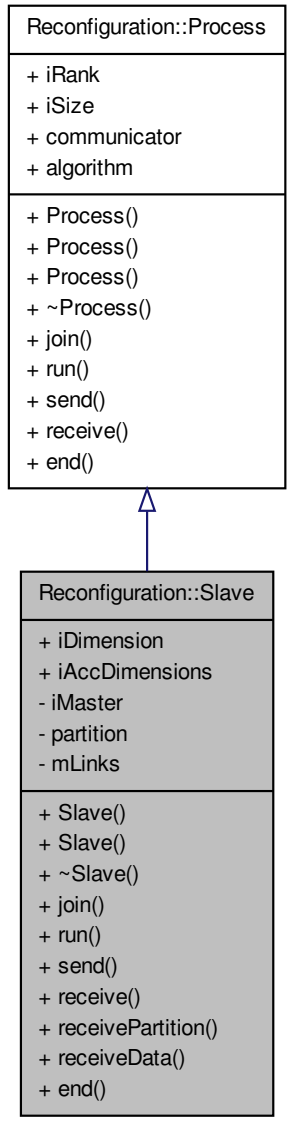
The documentation for this class was generated from the following files:

- confproblem.pb.h
- confproblem.pb.cc

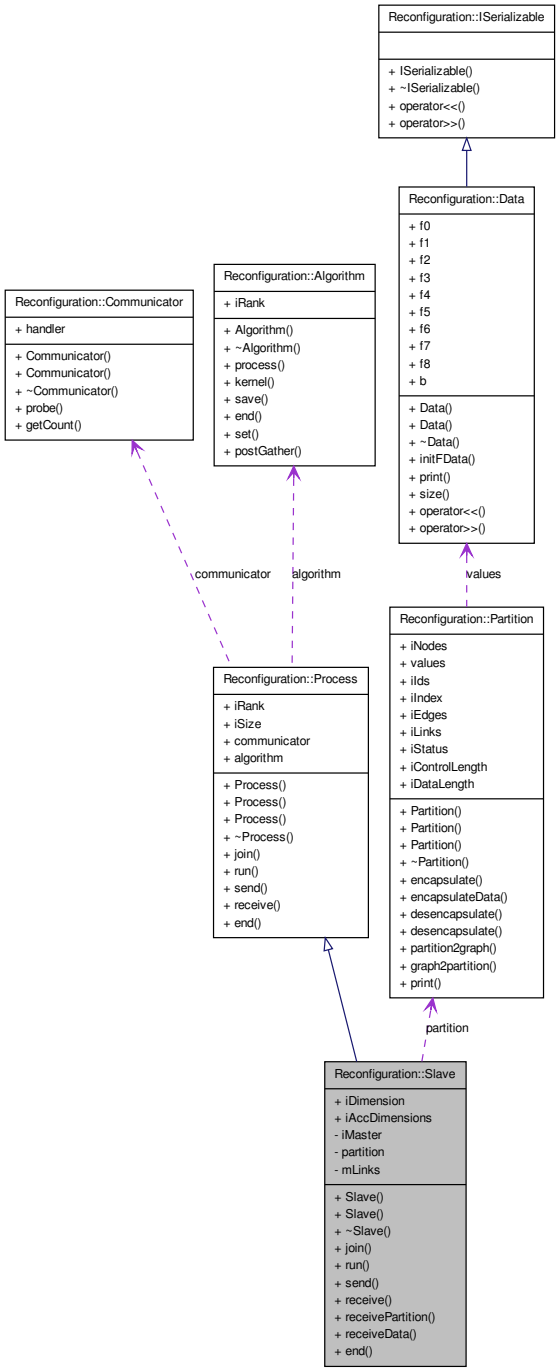
7.61 Reconfiguration::Slave Class Reference

`#include <Process.h>`

Inheritance diagram for Reconfiguration::Slave:



Collaboration diagram for Reconfiguration::Slave:



7.61.1 Public Member Functions

- Slave ()

  Constructor of the SuperMaster class.

- Slave (int, int, Algorithm ∗)

  Constructor of the Slave class.

- virtual ∼Slave ()

  virtual destructor of the class Slave.

- void join ()

  Protocol used by the Slave process to join the execution.

- void run ()

  Executes the user specified algorithm.

- void send (int)

  Not directly used.

- void receive ()

  Receives a new dataframe from its master process.

- void receivePartition (int)

  Receives a new partition dataframe from its master process.

- void receiveData (int)

  Receives a new data dataframe from its master process.

- void end ()

  Responds the finalization request sent by the master.

7.61.2 Public Attributes

- int iDimension

- int * iAccDimensions

7.61.3 Detailed Description

class Slave Slave process of the framework.

7.61.4 Constructor & Destructor Documentation

Reconfiguration::Slave::Slave ( )

Constructor of the SuperMaster class.

Returns

*this

Creates an object SuperMaster. Calls the father default constructor and sets the pointers to NULL.

Reconfiguration::Slave::Slave ([int]iRank, int *iMaster,* Algorithm * *algorithm*)

Constructor of the Slave class.

Parameters

in	*iRank*	Process rank.
in	*iMaster*	Father process rank.
in	*algorithm*	Pointer to the algorithm object to be solved.

Returns

*this

Creates an object Slave and initiates its communicator channel with the master process. Calls the father default constructor.

Reconfiguration::Slave::~Slave ( ) `[virtual]`

virtual destructor of the class Slave.

Returns

void

Destroys the Slave object.

7.61.5 Member Function Documentation

void Reconfiguration::Slave::end ( ) [virtual]

Responds the finalization request sent by the master.

Returns

void

Responds the finalization request sent by the master process and sends the TAG_END as answer. Changes the global flag value.

Exceptions

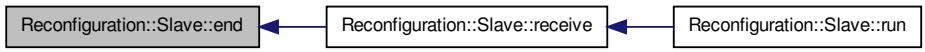
MPIException	Error on MPI layer.

See also

TAGS

Implements Reconfiguration::Process.

Here is the caller graph for this function:



void Reconfiguration::Slave::join ( ) [virtual]

Protocol used by the Slave process to join the execution.

Returns

void

The Slave process sends a request to the SuperMaster process with its hostname. If the request is accepted, the slave receives the problems configuration initiating then its links configuration structure.

Exceptions

MPIException	Error on MPI layer.
InternalException	No hostname file.

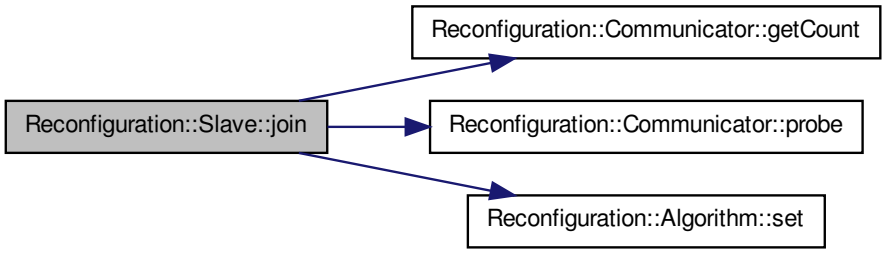
See also

mLinks

TAGS

Implements Reconfiguration::Process.

Here is the call graph for this function:



void Reconfiguration::Slave::receive ( ) `[virtual]`

Receives a new dataframe from its master process.

Returns

void

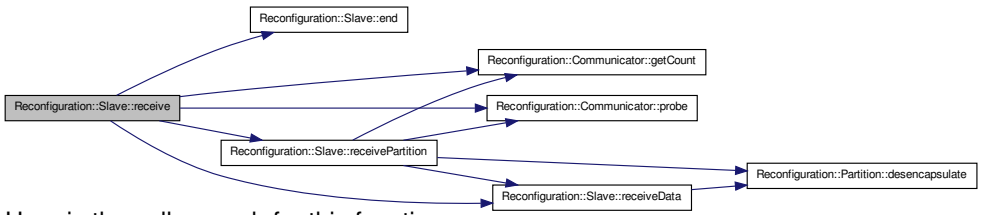
Waits until a new dataframe is received. Acts depending on the dataframe type received. The use of the function probe is in this case, mandatory.

Exceptions

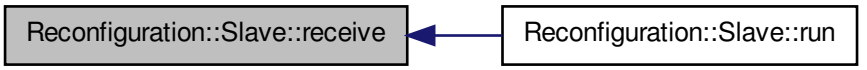
MPIException	Error on MPI layer.

Implements Reconfiguration::Process.

Here is the call graph for this function:



Here is the caller graph for this function:



void Reconfiguration::Slave::receiveData ([int]iSize)

Receives a new data dataframe from its master process.

Parameters

in	*iSize*	Size of the buffer that contains the dataframe.

Returns

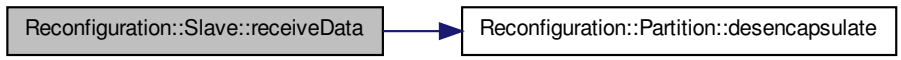
void

Receives the new data and desencapsulates it.

Exceptions

MPIException	Error on MPI layer.

Here is the call graph for this function:



Here is the caller graph for this function:

void Reconfiguration::Slave::receivePartition ([int]iSize)

Receives a new partition dataframe from its master process.

Parameters

in	*iSize*	Size of the buffer that contains the dataframe.

Returns

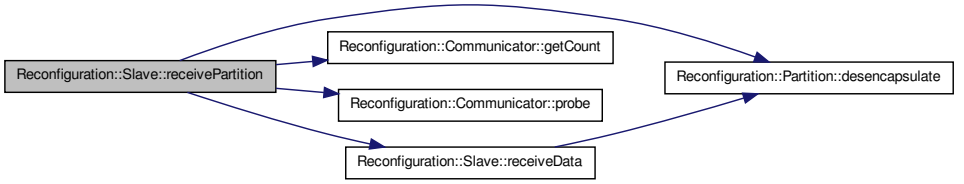
void

Receives the new partition and desencapsulates it. It also waits until a new data dataframe is received.

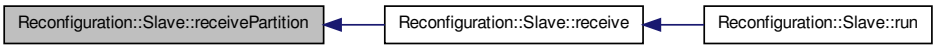
Exceptions

MPIException	Error on MPI layer.

Here is the call graph for this function:



Here is the caller graph for this function:



void Reconfiguration::Slave::run ( ) `[virtual]`

Executes the user specified algorithm.

Returns

void

The Slave process executes indefinitely the following steps, until it fulfills the ending criteria:

Receives a new partition from its master process.

Desencapsulates the partition into a new graph.

Executes by callback the algorithm kernel.

Encapsulates the partition and sends it back to its master. (TAG_RESULTS) When the ending criteria is fulfilled, the partition encapsulated with the TAG_FINALIZE tag is sent. MPIException Error on MPI layer.

See also

mLinks

TAGS

Returns

void

The Slave process executes indefinitely the following steps, until it fulfills the ending criteria:

Receives a new partition from its master process.

Desencapsulates the partition into a new graph.

Executes by callback the algorithm kernel.

Encapsulates the partition and sends it back to its master. (TAG_RESULTS) When the ending criteria is fulfilled, the partition encapsulated with the TAG_FINALIZE tag is sent.

Exceptions

MPIException	Error on MPI layer.

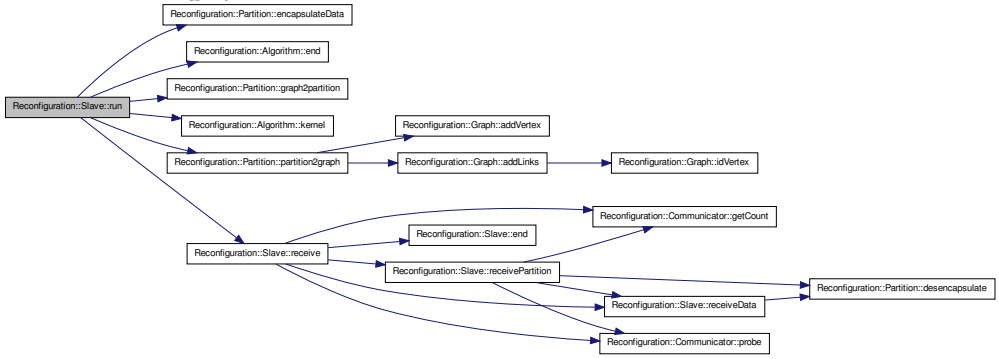
See also

mLinks

TAGS

Implements Reconfiguration::Process.

Here is the call graph for this function:



void Reconfiguration::Slave::send ([int]) `[virtual]`

Not directly used.

Returns

void

See also

Method run

Implements Reconfiguration::Process.

7.61.6 Member Data Documentation

int * Reconfiguration::Slave::iAccDimensions

iAccDimensions Array of accumulated length per problem dimension. The last position of this array must be equal to the number of vertices of the graph.

int Reconfiguration::Slave::iDimension

Problem dimension.

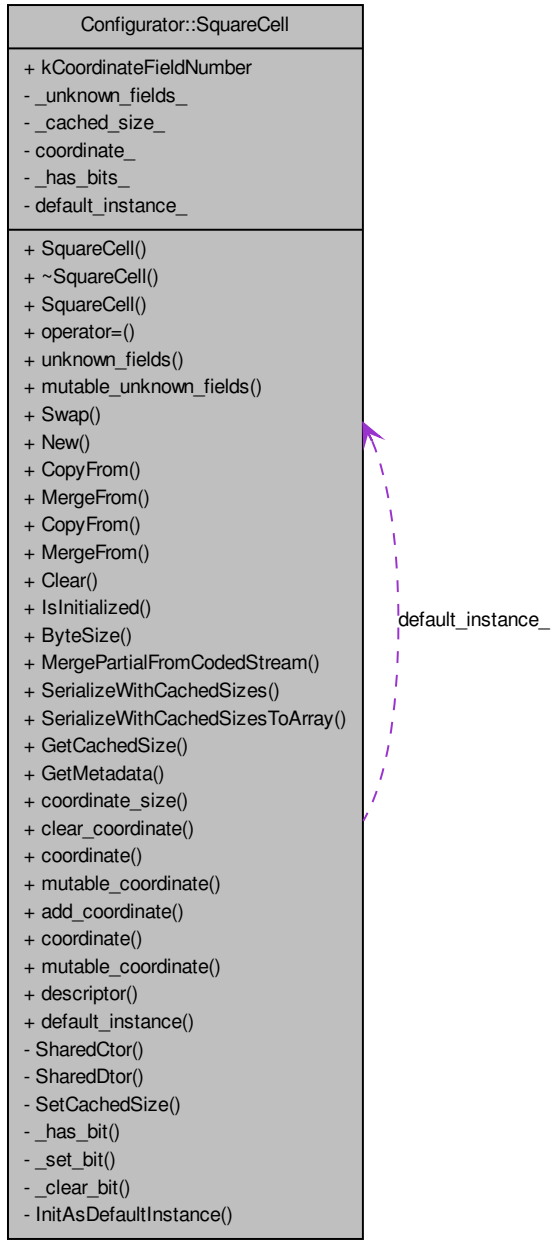
The documentation for this class was generated from the following files:

- Process.h
- Process.cpp

7.62 Configurator::SquareCell Class Reference

`#include <confproblem.pb.h>`

Collaboration diagram for Configurator::SquareCell:



7.62.1 Public Member Functions

- SquareCell ()
- virtual ∼SquareCell ()
- SquareCell (const SquareCell &from)
- SquareCell & operator= (const SquareCell &from)
- const ::google::protobuf::UnknownFieldSet & unknown_fields () const
- inline::google::protobuf::UnknownFieldSet ∗ mutable_unknown_fields ()
- void Swap (SquareCell ∗other)
- SquareCell ∗ New () const
- void CopyFrom (const ::google::protobuf::Message &from)
- void MergeFrom (const ::google::protobuf::Message &from)
- void CopyFrom (const SquareCell &from)
- void MergeFrom (const SquareCell &from)
- void Clear ()
- bool IsInitialized () const
- int ByteSize () const
- bool MergePartialFromCodedStream (::google::protobuf::io::CodedInputStream ∗input)
- void SerializeWithCachedSizes (::google::protobuf::io::CodedOutputStream ∗output) const
- ::google::protobuf::uint8 ∗ SerializeWithCachedSizesToArray (::google::protobuf::uint8 ∗output) const
- int GetCachedSize () const
- ::google::protobuf::Metadata GetMetadata () const
- int coordinate_size () const
- void clear_coordinate ()
- const ::Configurator::Coordinate & coordinate (int index) const
- inline::Configurator::Coordinate ∗ mutable_coordinate (int index)
- inline::Configurator::Coordinate ∗ add_coordinate ()
- const ::google::protobuf::RepeatedPtrField< ::Configurator::Coordinate > & coordinate () const
- inline::google::protobuf::RepeatedPtrField< ::Configurator::Coordinate > ∗ mutable_-coordinate ()

7.62.2 Static Public Member Functions

- static const ::google::protobuf::Descriptor * descriptor ()
- static const SquareCell & default_instance ()

7.62.3 Static Public Attributes

- static const int kCoordinateFieldNumber = 1

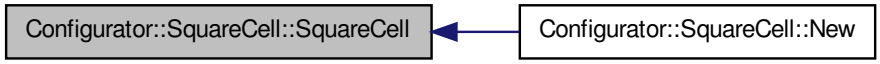
7.62.4 Friends

- void protobuf_AddDesc_confproblem_2eproto ()
- void protobuf_AssignDesc_confproblem_2eproto ()
- void protobuf_ShutdownFile_confproblem_2eproto ()

7.62.5 Constructor & Destructor Documentation

Configurator::SquareCell::SquareCell ( )

Here is the caller graph for this function:



Configurator::SquareCell::~SquareCell ( ) [virtual]

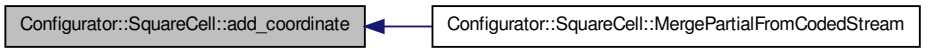
Configurator::SquareCell::SquareCell ([const SquareCell &]from)

Here is the call graph for this function:

7.62.6 Member Function Documentation

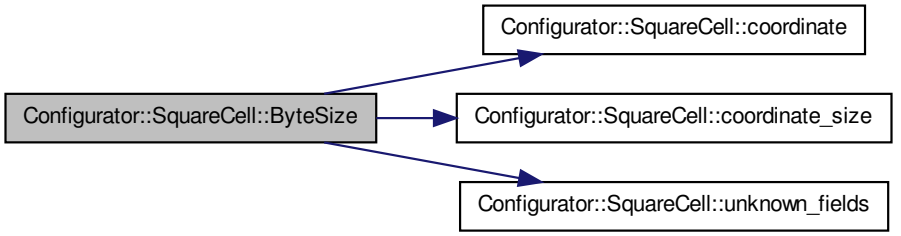
Configurator::Coordinate * Configurator::SquareCell::add_coordinate ( ) [inline]

Here is the caller graph for this function:



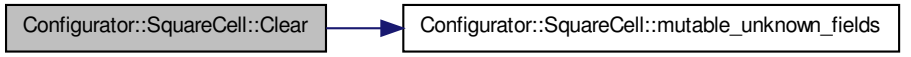
int Configurator::SquareCell::ByteSize ( ) const

Here is the call graph for this function:

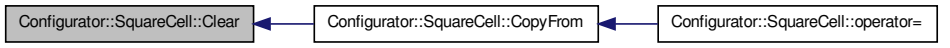


void Configurator::SquareCell::Clear ( )

Here is the call graph for this function:



Here is the caller graph for this function:

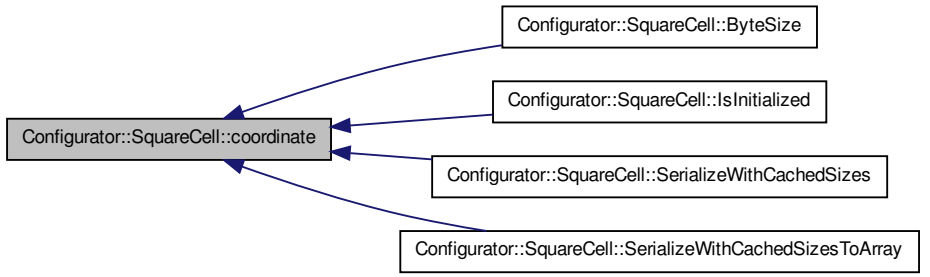


void Configurator::SquareCell::clear_coordinate ( ) `[inline]`

const ::Configurator::Coordinate & Configurator::SquareCell::coordinate ([int]index) const `[inline]`

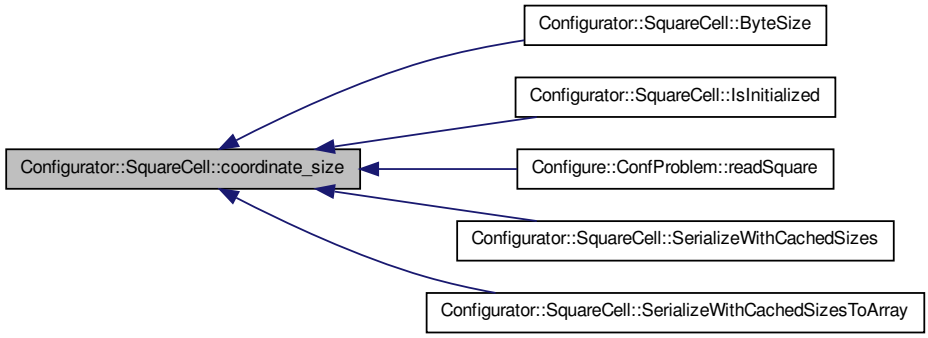
const ::google::protobuf::RepeatedPtrField<::Configurator::Coordinate > & Configurator::SquareCell::coordinate ( ) const `[inline]`

Here is the caller graph for this function:



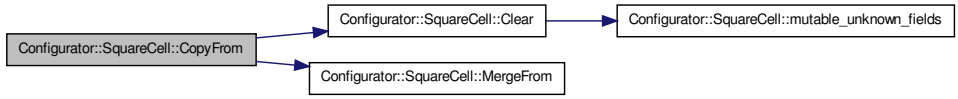
int Configurator::SquareCell::coordinate_size ( ) const [inline]

Here is the caller graph for this function:



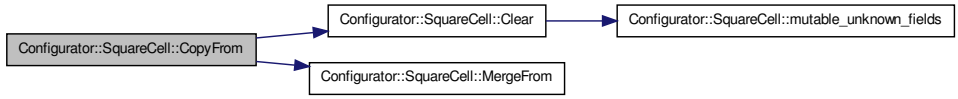
void Configurator::SquareCell::CopyFrom ([const SquareCell &]from)

Here is the call graph for this function:



void Configurator::SquareCell::CopyFrom ([const ::google::protobuf::Message &]from)

Here is the call graph for this function:



Here is the caller graph for this function:

const SquareCell & Configurator::SquareCell::default_instance ( ) [static]

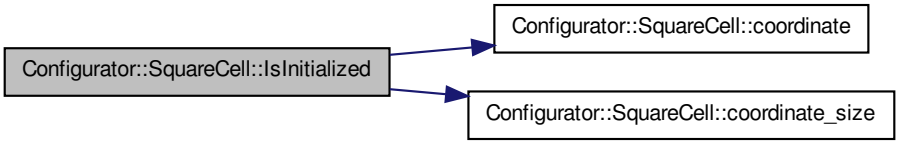
const ::google::protobuf::Descriptor * Configurator::SquareCell::descriptor ( ) [static]

int Configurator::SquareCell::GetCachedSize ( ) const [inline]

google::protobuf::Metadata Configurator::SquareCell::GetMetadata ( ) const

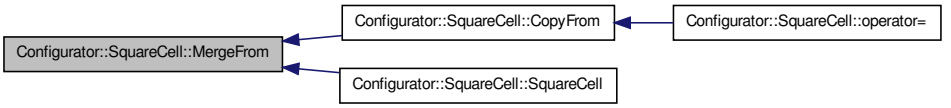
bool Configurator::SquareCell::IsInitialized ( ) const

Here is the call graph for this function:



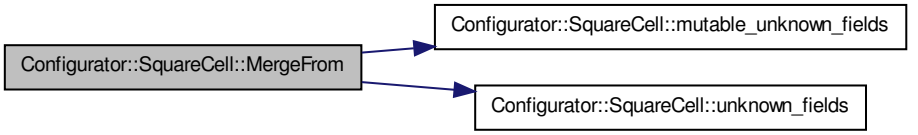
void Configurator::SquareCell::MergeFrom ([const ::google::protobuf::Message &]from)

Here is the caller graph for this function:



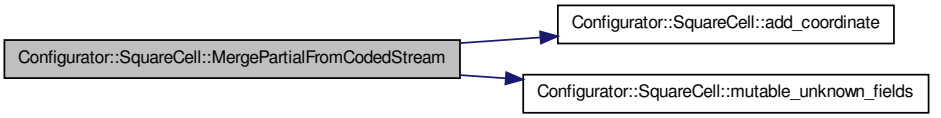
void Configurator::SquareCell::MergeFrom ([const SquareCell &]from)

Here is the call graph for this function:



bool Configurator::SquareCell::MergePartialFromCodedStream ([::google::protobuf::io::CodedInputStream ∗]input)

Here is the call graph for this function:

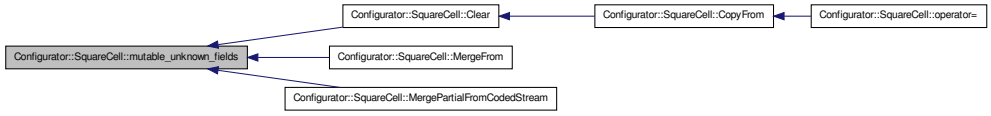


Configurator::Coordinate ∗ Configurator::SquareCell::mutable_coordinate ([int]index) `[inline]`

google::protobuf::RepeatedPtrField<::Configurator::Coordinate > ∗ Configurator::SquareCell::mutable_coordinate ( ) `[inline]`

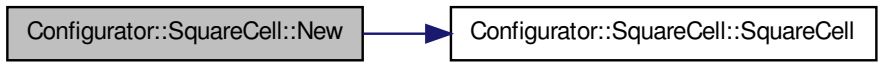
inline ::google::protobuf::UnknownFieldSet∗ Configurator::SquareCell::mutable_-unknown_fields ( ) `[inline]`

Here is the caller graph for this function:



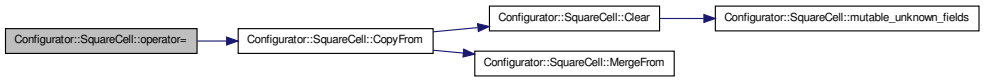
SquareCell * Configurator::SquareCell::New ( ) const

Here is the call graph for this function:



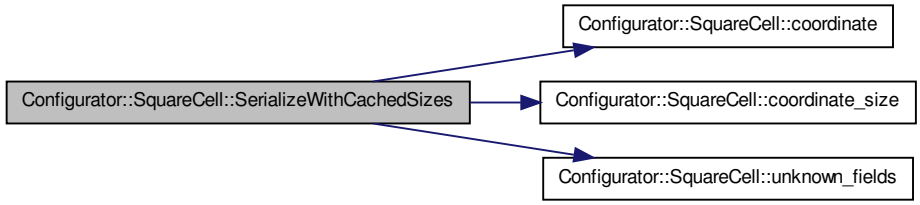
SquareCell& Configurator::SquareCell::operator= ([const SquareCell &]from) `[inline]`

Here is the call graph for this function:



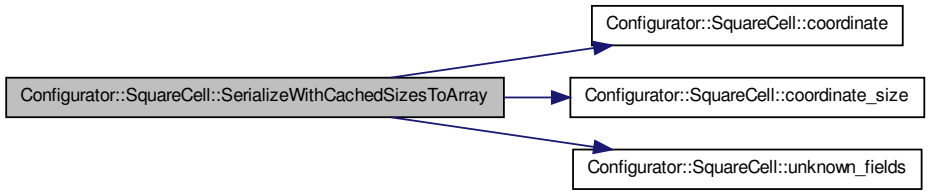
void Configurator::SquareCell::SerializeWithCachedSizes ([::google::protobuf::io::CodedOutputStream *]output) const

Here is the call graph for this function:



google::protobuf::uint8 * Configurator::SquareCell::SerializeWithCachedSizesToArray ([::google::protobuf::uint8 *]output) const

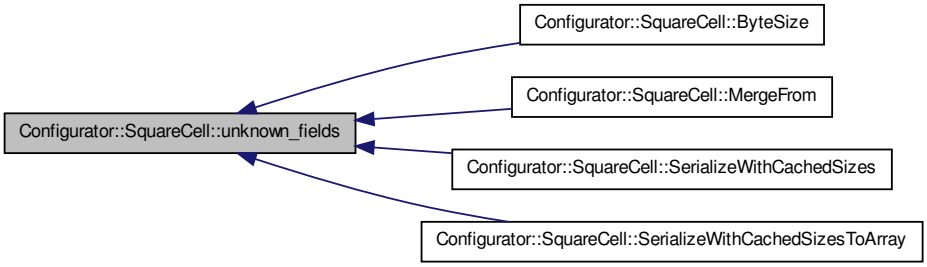
Here is the call graph for this function:



void Configurator::SquareCell::Swap ([SquareCell ∗]other)

const ::google::protobuf::UnknownFieldSet& Configurator::SquareCell::unknown_-fields ( ) const [inline]

Here is the caller graph for this function:



7.62.7 Friends And Related Function Documentation

void protobuf_AddDesc_confproblem_2eproto ( ) [friend]

void protobuf_AssignDesc_confproblem_2eproto ( ) [friend]

void protobuf_ShutdownFile_confproblem_2eproto ( ) [friend]

7.62.8 Member Data Documentation

const int Configurator::SquareCell::kCoordinateFieldNumber = 1 [static]

The documentation for this class was generated from the following files:

- confproblem.pb.h
- confproblem.pb.cc

7.63 Configurator::StaticDescriptorInitializer_confcluster_2eproto Struct Reference

7.63.1 Public Member Functions

- StaticDescriptorInitializer_confcluster_2eproto ()

7.63.2 Constructor & Destructor Documentation

Configurator::StaticDescriptorInitializer_confcluster_2eproto::StaticDescriptorInitializer_confcluster_2eproto ( )
`[inline]`

The documentation for this struct was generated from the following file:

- confcluster.pb.cc

7.64 Configurator::StaticDescriptorInitializer_confloadbalancer_2eproto Struct Reference

7.64.1 Public Member Functions

- StaticDescriptorInitializer_confloadbalancer_2eproto ()

7.64.2 Constructor & Destructor Documentation

Configurator::StaticDescriptorInitializer_confloadbalancer_2eproto::StaticDescriptorInitializer_confloadbalancer_2eproto ( ) `[inline]`

Here is the call graph for this function:

The documentation for this struct was generated from the following file:

- confloadbalancer.pb.cc

7.65 Configurator::StaticDescriptorInitializer_confpartitioner_2eproto Struct Reference

7.65.1 Public Member Functions

- StaticDescriptorInitializer_confpartitioner_2eproto ()

7.65.2 Constructor & Destructor Documentation

Configurator::StaticDescriptorInitializer_confpartitioner_2eproto::StaticDescriptorInitializer_confpartitioner_2eproto ( ) `[inline]`

Here is the call graph for this function:

The documentation for this struct was generated from the following file:

- confpartitioner.pb.cc

7.66 Configurator::StaticDescriptorInitializer_confproblem_2eproto Struct Reference

7.66.1 Public Member Functions

- StaticDescriptorInitializer_confproblem_2eproto ()

7.66.2 Constructor & Destructor Documentation

Configurator::StaticDescriptorInitializer_confproblem_2eproto::StaticDescriptorInitializer_confproblem_2eproto ( ) `[inline]`

Here is the call graph for this function:

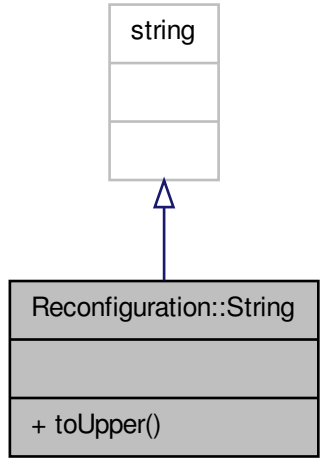
The documentation for this struct was generated from the following file:

- confproblem.pb.cc

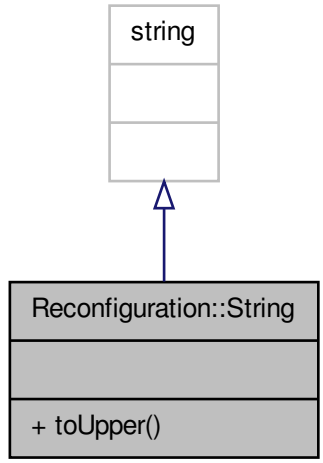
7.67 Reconfiguration::String Class Reference

`#include <String.h>`

Inheritance diagram for Reconfiguration::String:



Collaboration diagram for Reconfiguration::String:



7.67.1 Static Public Member Functions

- static char toUpper (char a)

  Converts a string to an UPPER string.

7.67.2 Detailed Description

Extends the features of the standard string class.

7.67.3 Member Function Documentation

static char Reconfiguration::String::toUpper ([char]a) `[inline, static]`

Converts a string to an UPPER string.

Returns

Converted string. Converts a string to an UPPER string, char by char.

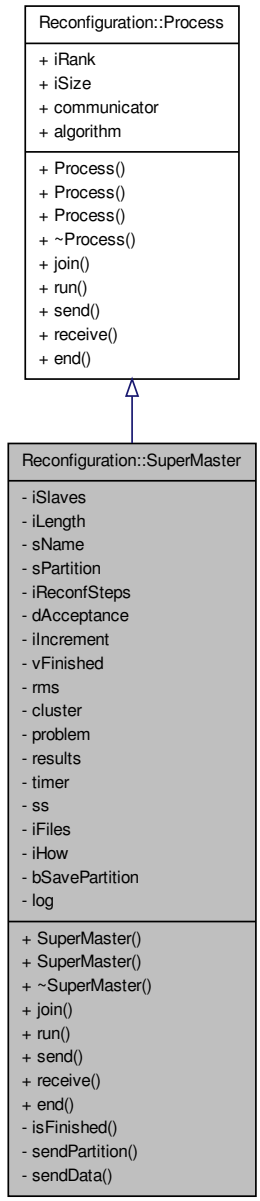
The documentation for this class was generated from the following file:

- String.h

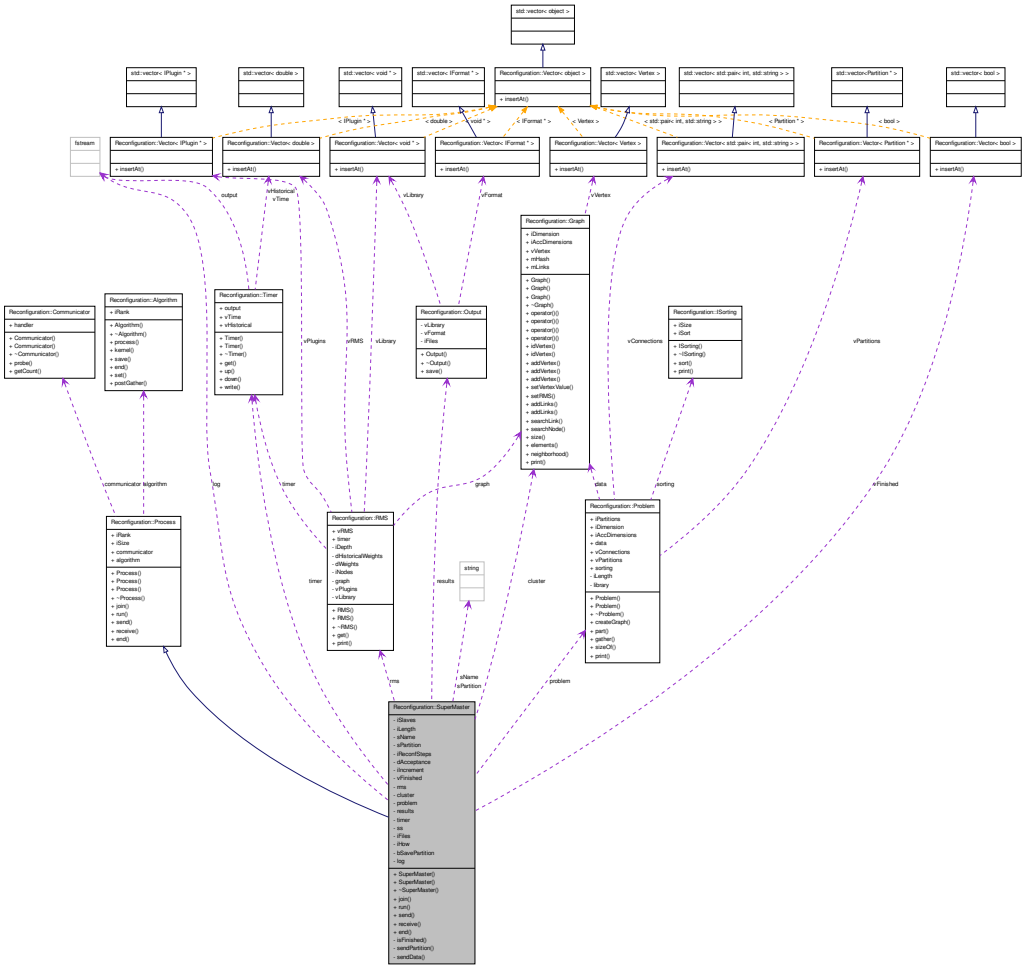
7.68 Reconfiguration::SuperMaster Class Reference

`#include <Process.h>`

Inheritance diagram for Reconfiguration::SuperMaster:



Collaboration diagram for Reconfiguration::SuperMaster:



7.68.1 Public Member Functions

- SuperMaster ()

  Constructor of the SuperMaster class.

- SuperMaster (char **, int, Algorithm *)

  Constructor of the SuperMaster class.

- virtual ~SuperMaster ()

  virtual destructor of the class SuperMaster.

- void join ()

  Protocol used by the SuperMaster process to join the execution.

- void run ()

  Calculates the new RMS values and sends the PARTITION or the DATA dataframe (depending on the heuristic.conf configuration file) to the slaves and manages the timers. Receives from the slaves.

- void send (int)

  Sends a PARTITION or DATA dataframe to the slaves.

- void receive ()

  Receives dataframes from the slaves.

- void end ()

  Sends all processes the TAG_FINALIZE signal.

7.68.2 Detailed Description

class SuperMaster Master process of the framework.

7.68.3 Constructor & Destructor Documentation

Reconfiguration::SuperMaster::SuperMaster ( )

Constructor of the SuperMaster class.

Returns

∗this

Creates an object SuperMaster. Calls the father default constructor and sets the pointers to NULL.

Reconfiguration::SuperMaster::SuperMaster ([char ∗∗]args, int *iRank*, Algorithm ∗ *algorithm*)

Constructor of the SuperMaster class.

Parameters

in	*args*	Input arguments.
in	*iRank*	MPI Rank of the process.
in	*algorithm*	Algorithm object to be solved.

Returns

∗this

Creates an object SuperMaster. Calls the father default constructor. This process reads the configuration files and mainly initiates the system graph, the application graph, the timers and the output interface. The SuperMaster process waits for the reception of the slave hostname, that has joined the execution.

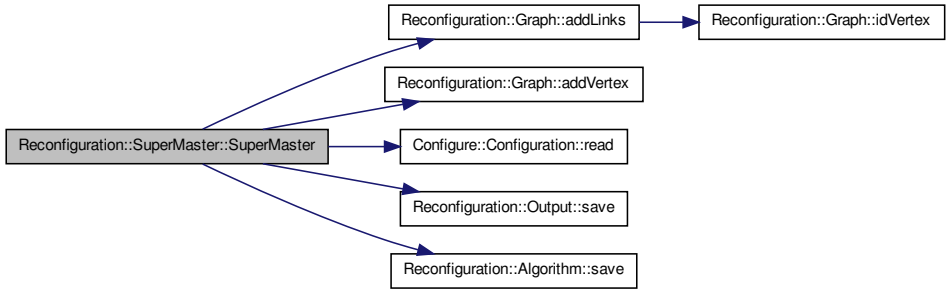
Exceptions

MPIException	Error on MPI layer.
ConfigurationException	Error on configuration file.

See also

SUPERMASTER

Here is the call graph for this function:



Reconfiguration::SuperMaster::∼SuperMaster ( ) [virtual]

virtual destructor of the class SuperMaster.

Returns

void

Destroys the SuperMaster object.

7.68.4 Member Function Documentation

void Reconfiguration::SuperMaster::end ( ) [virtual]

Sends all processes the TAG_FINALIZE signal.

Returns

void

Sends to all processes the TAG_FINALIZE dataframe and waits until receiving as much TAG_END dataframes as slaves are, to close the communication chanel.

Exceptions

MPIException	Error on MPI layer.

See also

TAGS

Implements Reconfiguration::Process.

void Reconfiguration::SuperMaster::join ( ) [virtual]

Protocol used by the SuperMaster process to join the execution.

Returns

void

The SuperMaster sends the links configuration (read from the user config file) to all slaves, from those who has received the join request.

Exceptions

MPIException	Error on MPI layer.

See also

mLinks

TAGS

Implements Reconfiguration::Process.

void Reconfiguration::SuperMaster::receive ( ) [virtual]

Receives dataframes from the slaves.

Returns

void

Receive as many dataframes as slaves are involved. Depending on the tag received, we have to desencapsulate the information received. The use of the probe function is in this case mandatory. MPIException Error on MPI layer.

Returns

void

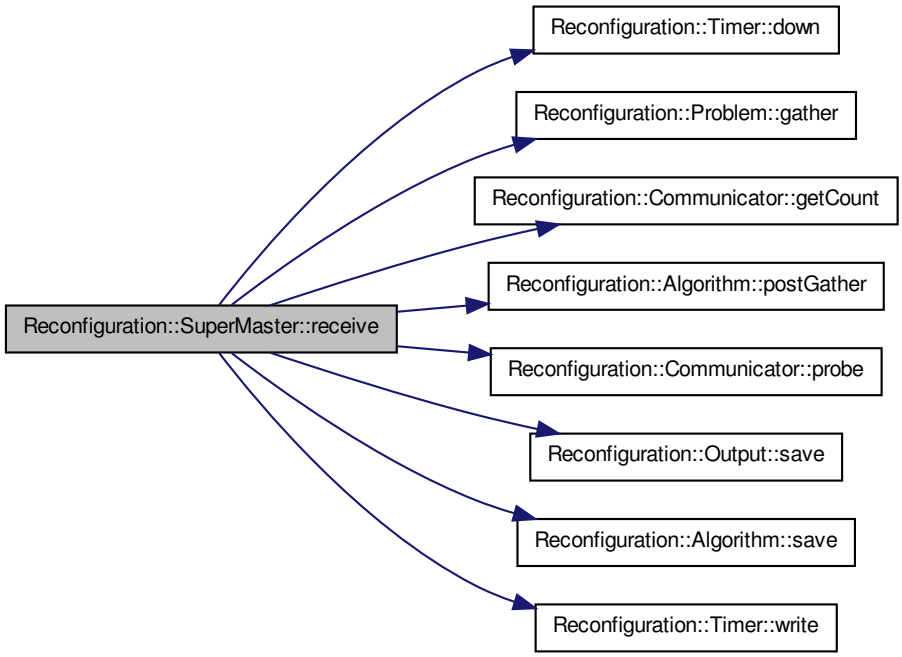
Receive as many dataframes as slaves are involved. Depending on the tag received, we have to desencapsulate the information received. The use of the probe function is in this case mandatory.

Exceptions

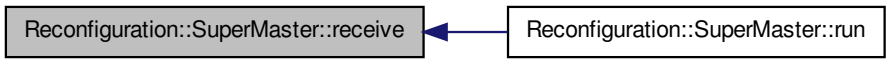
MPIException	Error on MPI layer.

Implements Reconfiguration::Process.

Here is the call graph for this function:



Here is the caller graph for this function:



void Reconfiguration::SuperMaster::run ( ) [virtual]

Calculates the new RMS values and sends the PARTITION or the DATA dataframe (depending on the heuristic.conf configuration file) to the slaves and manages the timers. Receives from the slaves.

Returns

void

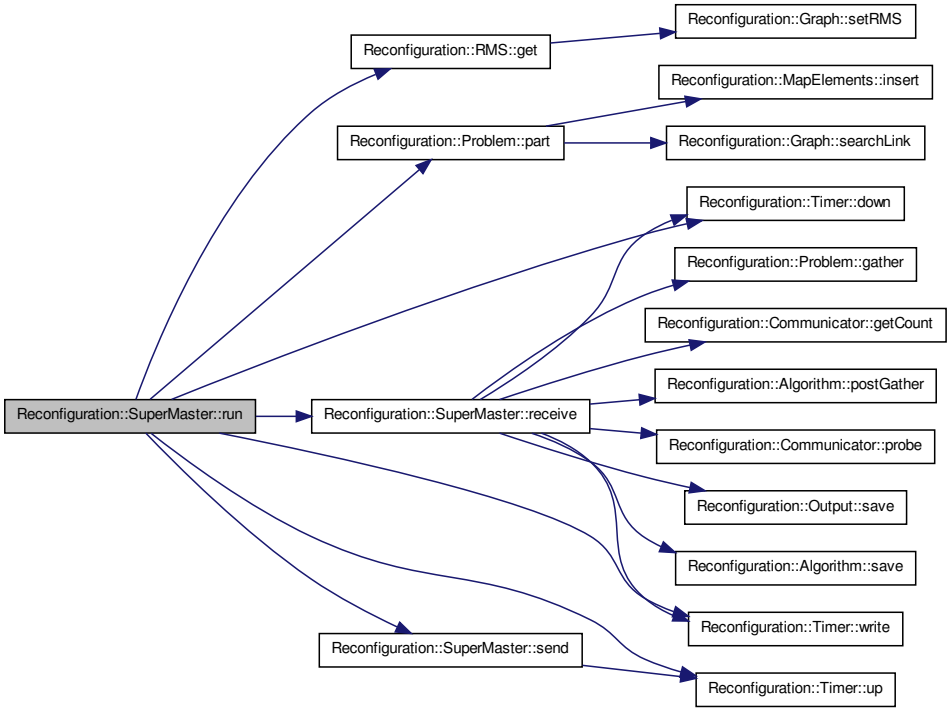
Calculates the new RMS values and sends the PARTITION or the DATA dataframe (depending on the heuristic.conf configuration file) to the slaves and manages the timers. Receives from the slaves to act under its conderation.

See also

DATAFRAME

Implements Reconfiguration::Process.

Here is the call graph for this function:



void Reconfiguration::SuperMaster::send ([int]iSend) `[virtual]`

Sends a PARTITION or DATA dataframe to the slaves.

Parameters

`in`	*iSend*	Tag to send.

Returns

void

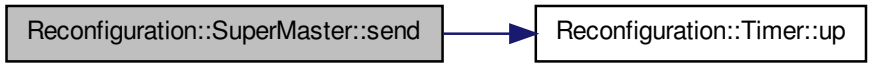
Sends a PARTITION or DATA dataframe to all the slaves that has not yet end the execution. If the dataframe is a PARTITION dataframe, not only the partition but also the data is encapsulated and sent to each slave. Otherwise if the dataframe is a DATA dataframe only the data is encapsulated. The timer count starts on each send operation.

See also

DATAFRAME

Implements Reconfiguration::Process.

Here is the call graph for this function:



Here is the caller graph for this function:

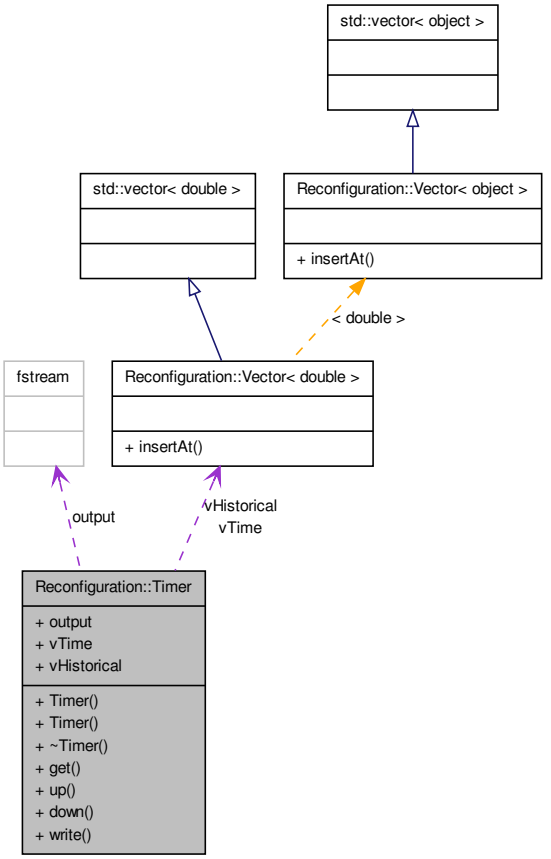
The documentation for this class was generated from the following files:

- Process.h
- Process.cpp

7.69 Reconfiguration::Timer Class Reference

`#include <Timer.h>`

Collaboration diagram for Reconfiguration::Timer:



7.69.1 Public Member Functions

- Timer ()

  Constructor of the Timer class.

- Timer (const char *, int, int)

  Constructor of the Timer class.

- virtual ~Timer ()

  virtual destructor of the class Timer.

- double get (int)

  Takes the current time in milliseconds and calculates the elapsed time.

- void up (int)

  Starts a certain timer.

- void down (int)

  Stops a certain timer.

- void write (int)

  Writes a certain timer result to a file.

7.69.2 Public Attributes

- std::fstream output
- Vector< double > vTime
- Vector< double > vHistorical

7.69.3 Detailed Description

Platform timer clock.

7.69.4 Constructor & Destructor Documentation

Reconfiguration::Timer::Timer ( )

Constructor of the Timer class.

Returns

∗this

Creates an object Timer.

Reconfiguration::Timer::Timer ([const char ∗]cFilename, int *iTimers,* int *iMaxDepth*)

Constructor of the Timer class.

Parameters

in	*cFilename*	Time filename.
in	*iTimers*	Number of timers (= number of slaves).
in	*iMaxDepth*	Number of historical values.

Returns

∗this

Creates an object Timer.

Reconfiguration::Timer::∼Timer ( ) `[virtual]`

virtual destructor of the class Timer.

Returns

void

Destroys the Timer object.

7.69.5 Member Function Documentation

void Reconfiguration::Timer::down ([int]iIndex)

Stops a certain timer.

Parameters

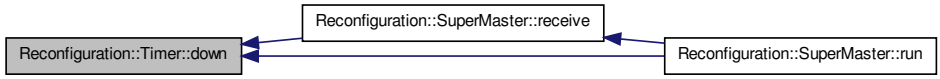
in	*iIndex*	Position of the array in which we must write the current time.

Returns

void

Stops a certain timer getting the actual timestamp and calculating the elapsed time.

Here is the caller graph for this function:



double Reconfiguration::Timer::get ([int]iIndex)

Takes the current time in milliseconds and calculates the elapsed time.

Parameters

in	*iIndex*	Position of the array in which we must write the current time.

Returns

double

Gets the current time in milliseconds. If this function was previously called, the elapsed time will be calculated. Returns the elapsed time.

void Reconfiguration::Timer::up ([int]iIndex)

Starts a certain timer.

Parameters

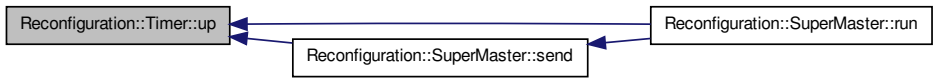
in	*iIndex*	Position of the array in which we must write the current time.

Returns

void

Starts a certain timer getting the actual timestamp.

Here is the caller graph for this function:



void Reconfiguration::Timer::write ([int]iIndex)

Writes a certain timer result to a file.

Parameters

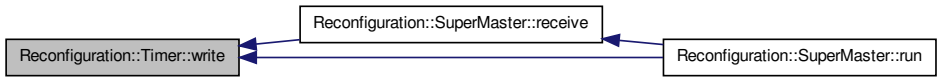
in	*iIndex*	Position of the array in which we must write the current time.

Returns

void

Writes the elapsed time of one of the timer to a file.

Here is the caller graph for this function:



7.69.6 Member Data Documentation

Reconfiguration::Timer::output

Handler of the time output file.

Reconfiguration::Timer::vHistorical

Time historical values.

Reconfiguration::Timer::vTime

Elapsed time of each process.

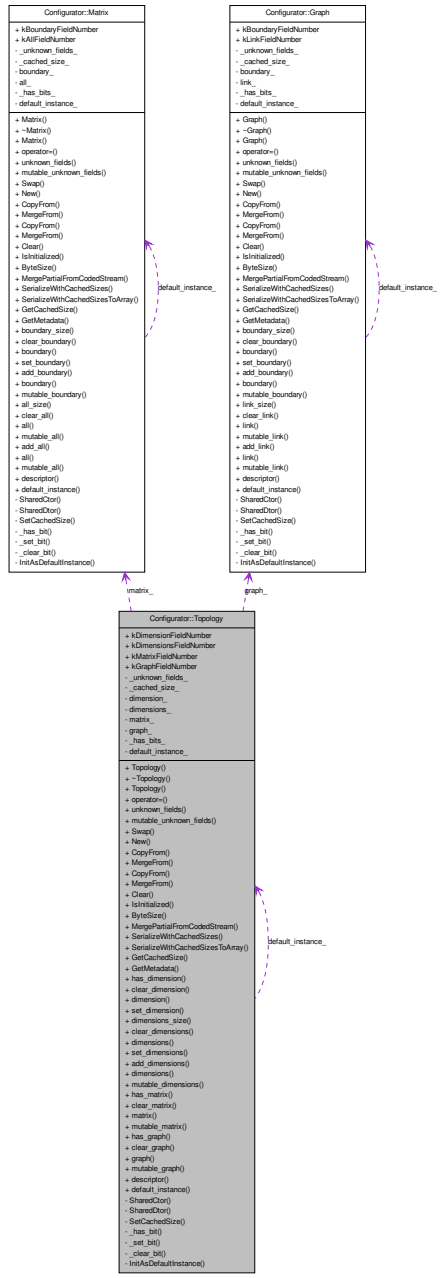
The documentation for this class was generated from the following files:

- Timer.h
- Timer.cpp

7.70 Configurator::Topology Class Reference

`#include <confproblem.pb.h>`

Collaboration diagram for Configurator::Topology:



7.70.1 Public Member Functions

- Topology ()
- virtual ∼Topology ()
- Topology (const Topology &from)
- Topology & operator= (const Topology &from)
- const ::google::protobuf::UnknownFieldSet & unknown_fields () const
- inline::google::protobuf::UnknownFieldSet ∗ mutable_unknown_fields ()
- void Swap (Topology ∗other)
- Topology ∗ New () const
- void CopyFrom (const ::google::protobuf::Message &from)
- void MergeFrom (const ::google::protobuf::Message &from)
- void CopyFrom (const Topology &from)
- void MergeFrom (const Topology &from)
- void Clear ()
- bool IsInitialized () const
- int ByteSize () const
- bool MergePartialFromCodedStream (::google::protobuf::io::CodedInputStream ∗input)
- void SerializeWithCachedSizes (::google::protobuf::io::CodedOutputStream ∗output) const
- ::google::protobuf::uint8 ∗ SerializeWithCachedSizesToArray (::google::protobuf::uint8 ∗output) const
- int GetCachedSize () const
- ::google::protobuf::Metadata GetMetadata () const
- bool has_dimension () const
- void clear_dimension ()
- inline::google::protobuf::int32 dimension () const
- void set_dimension (::google::protobuf::int32 value)
- int dimensions_size () const
- void clear_dimensions ()
- inline::google::protobuf::int32 dimensions (int index) const
- void set_dimensions (int index,::google::protobuf::int32 value)
- void add_dimensions (::google::protobuf::int32 value)
- const ::google::protobuf::RepeatedField< ::google::protobuf::int32 > & dimensions ()

const

- inline::google::protobuf::RepeatedField< ::google::protobuf::int32 > * mutable_dimensions ()
- bool has_matrix () const
- void clear_matrix ()
- const ::Configurator::Matrix & matrix () const
- inline::Configurator::Matrix * mutable_matrix ()
- bool has_graph () const
- void clear_graph ()
- const ::Configurator::Graph & graph () const
- inline::Configurator::Graph * mutable_graph ()

7.70.2 Static Public Member Functions

- static const ::google::protobuf::Descriptor * descriptor ()
- static const Topology & default_instance ()

7.70.3 Static Public Attributes

- static const int kDimensionFieldNumber = 1
- static const int kDimensionsFieldNumber = 2
- static const int kMatrixFieldNumber = 3
- static const int kGraphFieldNumber = 4

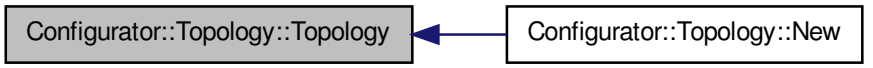
7.70.4 Friends

- void protobuf_AddDesc_confproblem_2eproto ()
- void protobuf_AssignDesc_confproblem_2eproto ()
- void protobuf_ShutdownFile_confproblem_2eproto ()

7.70.5 Constructor & Destructor Documentation

Configurator::Topology::Topology ( )

Here is the caller graph for this function:



Configurator::Topology::~Topology ( ) [virtual]

Configurator::Topology::Topology ([const Topology &]from)

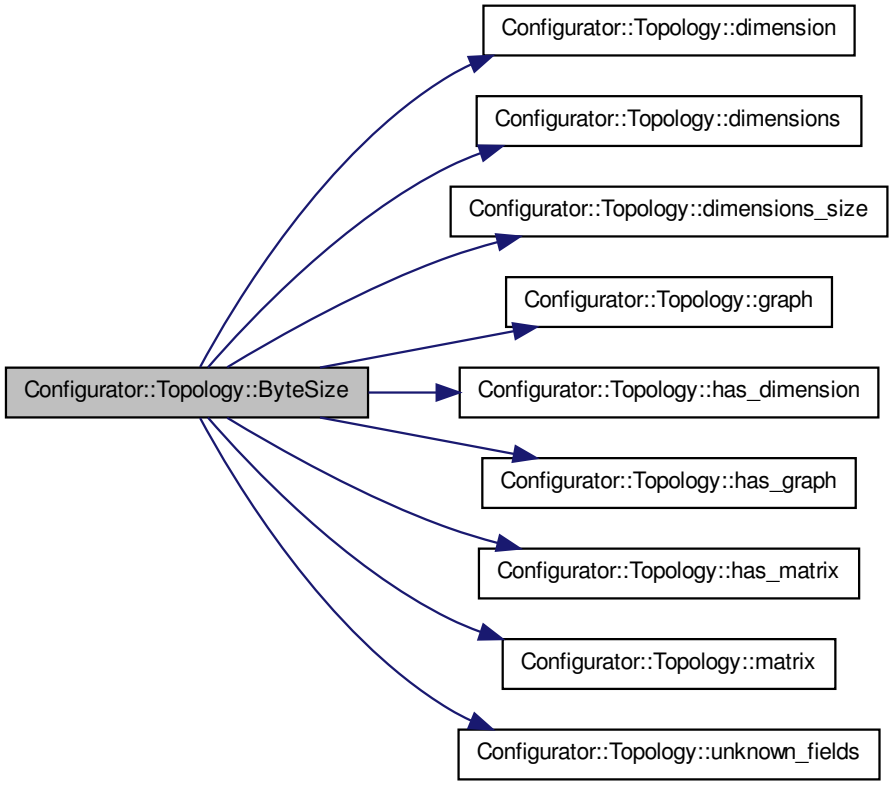
Here is the call graph for this function:

7.70.6 Member Function Documentation

void Configurator::Topology::add_dimensions ([::google::protobuf::int32]value) [inline]

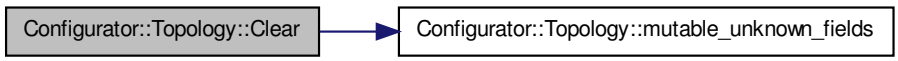
int Configurator::Topology::ByteSize ( ) const

Here is the call graph for this function:

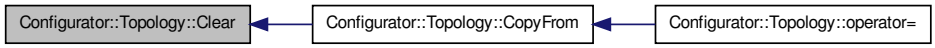


void Configurator::Topology::Clear ( )

Here is the call graph for this function:



Here is the caller graph for this function:



void Configurator::Topology::clear_dimension ( ) [inline]

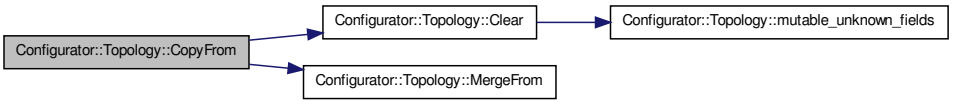
void Configurator::Topology::clear_dimensions ( ) [inline]

void Configurator::Topology::clear_graph ( ) [inline]

void Configurator::Topology::clear_matrix ( ) [inline]

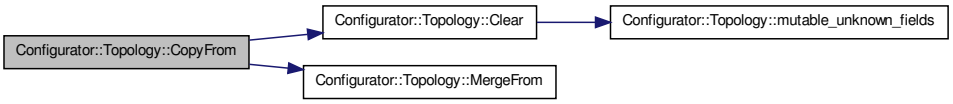
void Configurator::Topology::CopyFrom ([const Topology &]from)

Here is the call graph for this function:



void Configurator::Topology::CopyFrom ([const ::google::protobuf::Message &]from)

Here is the call graph for this function:



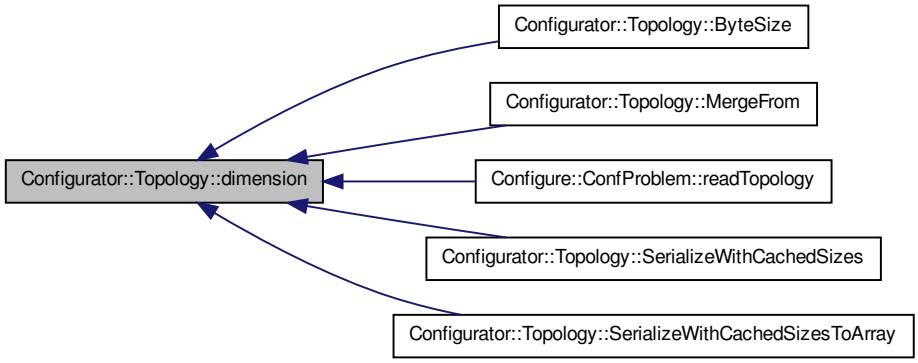
Here is the caller graph for this function:

const Topology & Configurator::Topology::default_instance ( ) [static]

const ::google::protobuf::Descriptor * Configurator::Topology::descriptor ( ) [static]

google::protobuf::int32 Configurator::Topology::dimension ( ) const [inline]

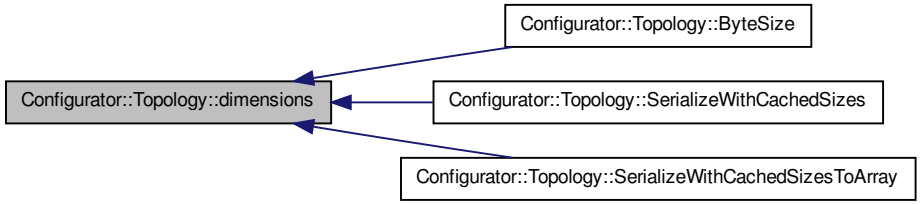
Here is the caller graph for this function:



google::protobuf::int32 Configurator::Topology::dimensions ([int]index) const `[inline]`

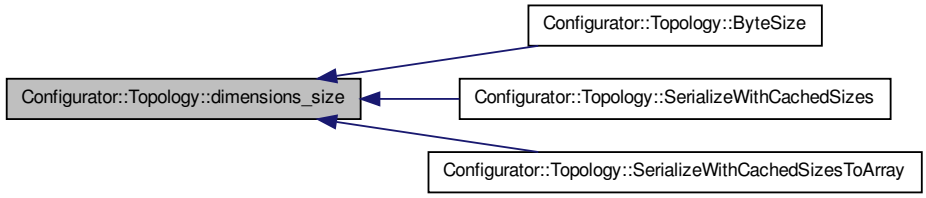
const ::google::protobuf::RepeatedField<::google::protobuf::int32 > & Configurator::Topology::dimensions ( ) const `[inline]`

Here is the caller graph for this function:



int Configurator::Topology::dimensions_size ( ) const `[inline]`

Here is the caller graph for this function:

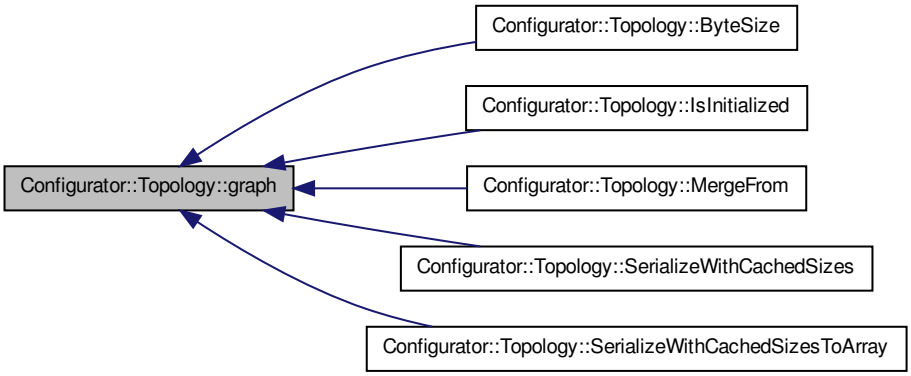


int Configurator::Topology::GetCachedSize ( ) const `[inline]`

google::protobuf::Metadata Configurator::Topology::GetMetadata ( ) const

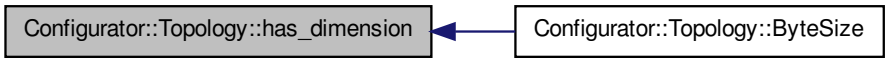
const ::Configurator::Graph & Configurator::Topology::graph ( ) const `[inline]`

Here is the caller graph for this function:



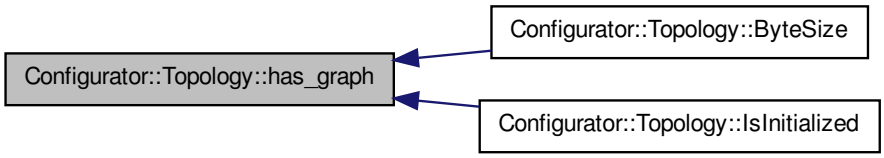
bool Configurator::Topology::has_dimension ( ) const [inline]

Here is the caller graph for this function:



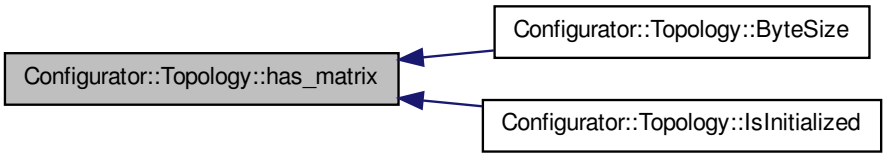
bool Configurator::Topology::has_graph ( ) const [inline]

Here is the caller graph for this function:



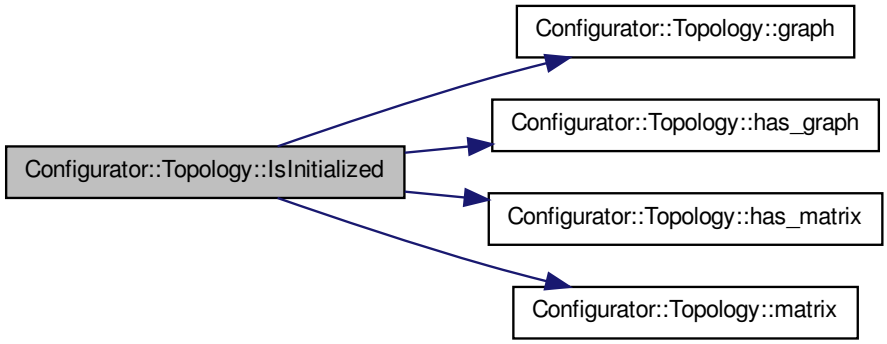
bool Configurator::Topology::has_matrix ( ) const [inline]

Here is the caller graph for this function:



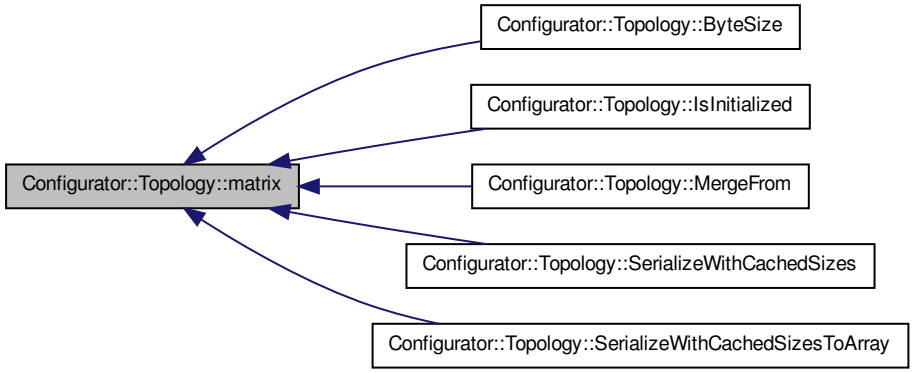
bool Configurator::Topology::IsInitialized ( ) const

Here is the call graph for this function:



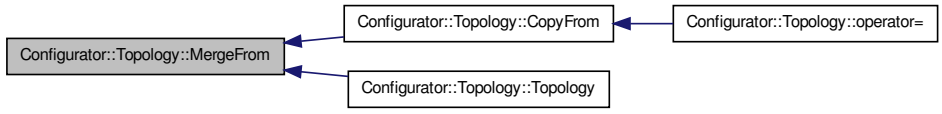
const ::Configurator::Matrix & Configurator::Topology::matrix ( ) const [inline]

Here is the caller graph for this function:



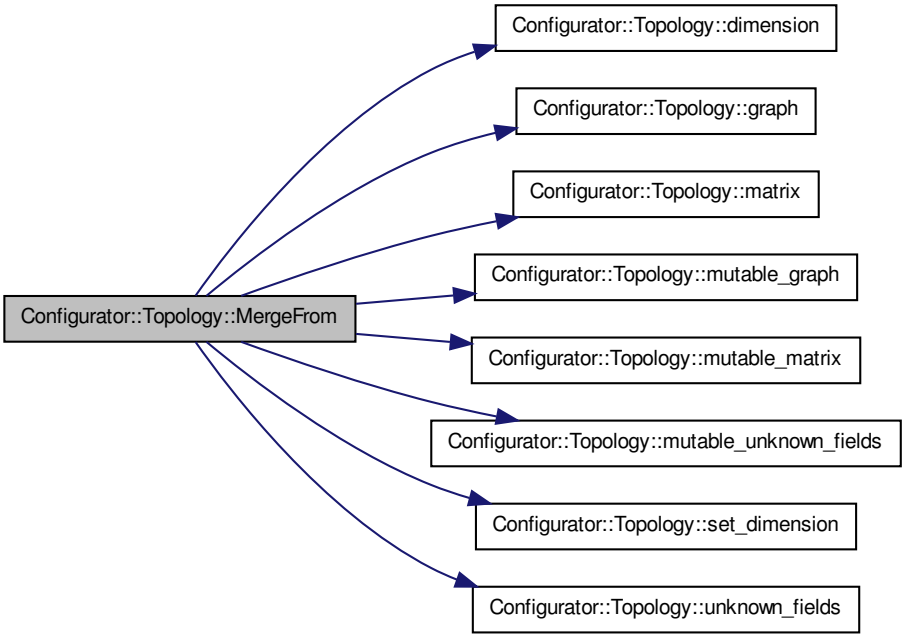
void Configurator::Topology::MergeFrom ([const ::google::protobuf::Message &]from)

Here is the caller graph for this function:



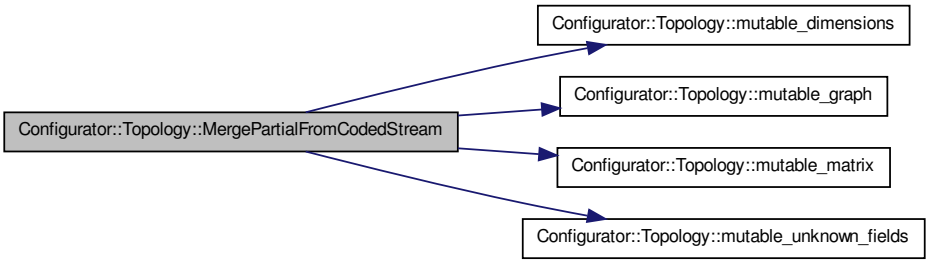
void Configurator::Topology::MergeFrom ([const Topology &]from)

Here is the call graph for this function:



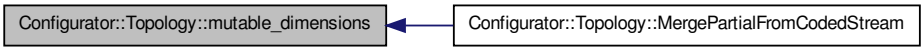
bool Configurator::Topology::MergePartialFromCodedStream ([::google::protobuf::io::CodedInputStream *]input)

Here is the call graph for this function:



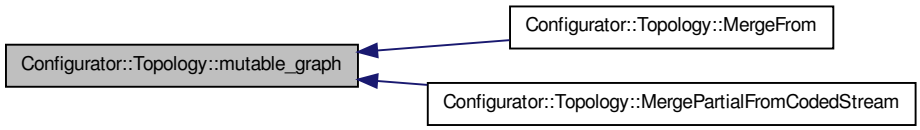
google::protobuf::RepeatedField<::google::protobuf::int32 > * Configurator::Topology::mutable_dimensions ( ) [inline]

Here is the caller graph for this function:



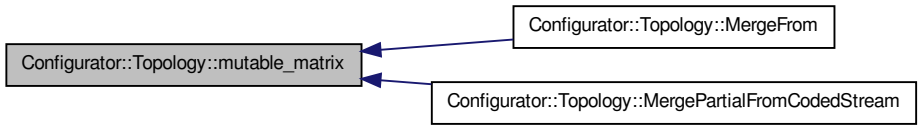
Configurator::Graph * Configurator::Topology::mutable_graph ( ) [inline]

Here is the caller graph for this function:



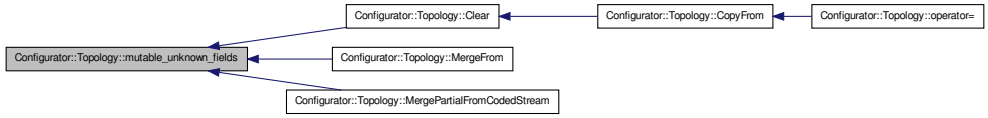
Configurator::Matrix * Configurator::Topology::mutable_matrix ( ) [inline]

Here is the caller graph for this function:



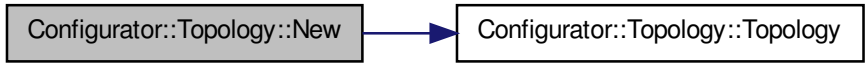
inline ::google::protobuf::UnknownFieldSet* Configurator::Topology::mutable_unknown_fields ( ) [inline]

Here is the caller graph for this function:



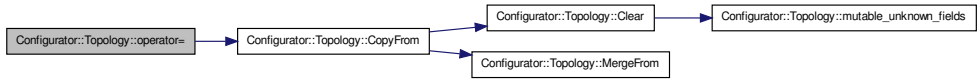
Topology * Configurator::Topology::New ( ) const

Here is the call graph for this function:



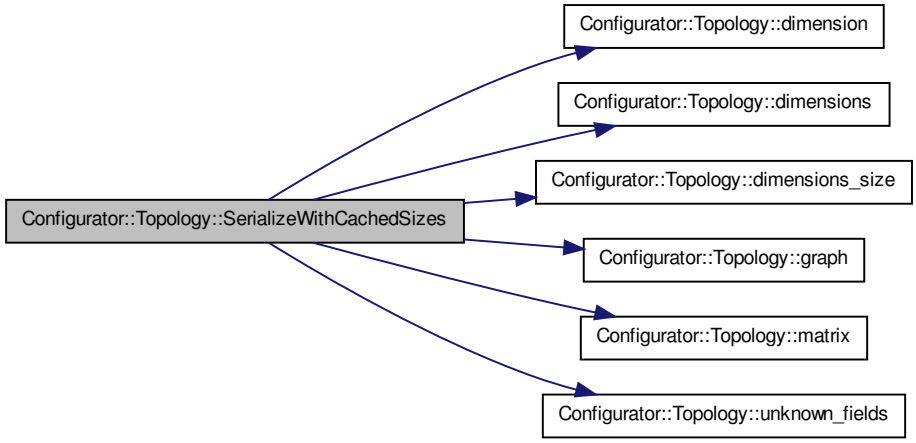
Topology& Configurator::Topology::operator= ([const Topology &]from) [inline]

Here is the call graph for this function:



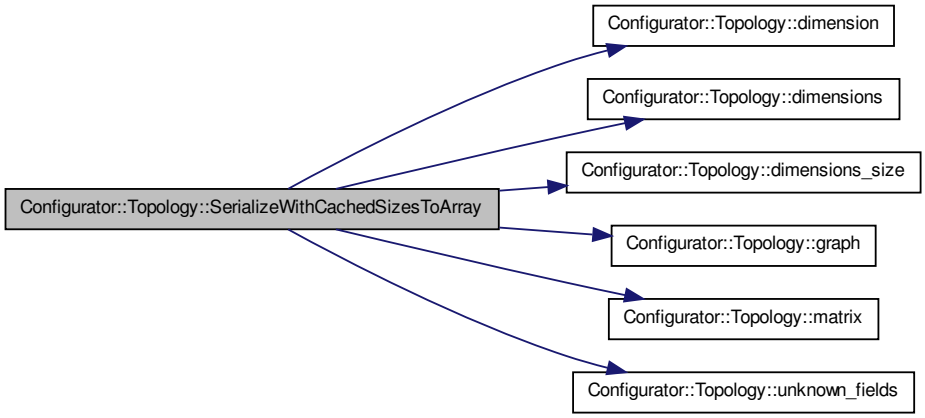
void Configurator::Topology::SerializeWithCachedSizes ([::google::protobuf::io::CodedOutputStream *]output) const

Here is the call graph for this function:



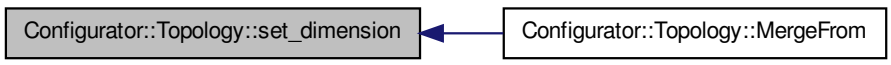
google::protobuf::uint8 * Configurator::Topology::SerializeWithCachedSizesToArray ([::google::protobuf::uint8 *]output) const

Here is the call graph for this function:



void Configurator::Topology::set_dimension ([::google::protobuf::int32]value) `[inline]`

Here is the caller graph for this function:

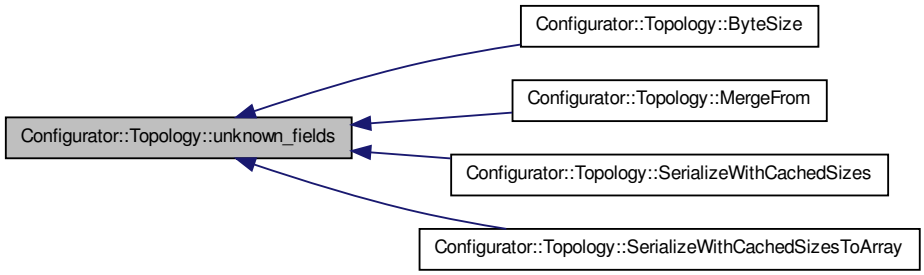


void Configurator::Topology::set_dimensions ([int]index, ::google::protobuf::int32 *value*) [inline]

void Configurator::Topology::Swap ([Topology ∗]other)

const ::google::protobuf::UnknownFieldSet& Configurator::Topology::unknown_-fields ( ) const [inline]

Here is the caller graph for this function:



7.70.7 Friends And Related Function Documentation

void protobuf_AddDesc_confproblem_2eproto ( ) [friend]

void protobuf_AssignDesc_confproblem_2eproto ( ) [friend]

void protobuf_ShutdownFile_confproblem_2eproto ( ) [friend]

7.70.8 Member Data Documentation

const int Configurator::Topology::kDimensionFieldNumber = 1 [static]

const int Configurator::Topology::kDimensionsFieldNumber = 2 [static]

const int Configurator::Topology::kGraphFieldNumber = 4 [static]

const int Configurator::Topology::kMatrixFieldNumber = 3 [static]

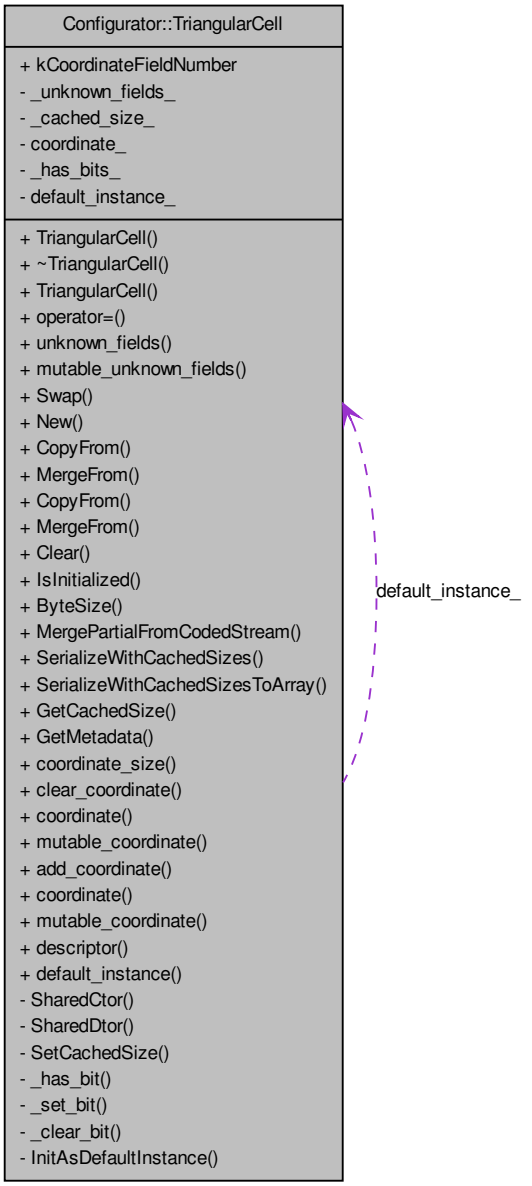
The documentation for this class was generated from the following files:

- confproblem.pb.h
- confproblem.pb.cc

7.71 Configurator::TriangularCell Class Reference

`#include <confproblem.pb.h>`

Collaboration diagram for Configurator::TriangularCell:



7.71.1 Public Member Functions

- TriangularCell ()
- virtual ~TriangularCell ()
- TriangularCell (const TriangularCell &from)
- TriangularCell & operator= (const TriangularCell &from)
- const ::google::protobuf::UnknownFieldSet & unknown_fields () const
- inline::google::protobuf::UnknownFieldSet * mutable_unknown_fields ()
- void Swap (TriangularCell *other)
- TriangularCell * New () const
- void CopyFrom (const ::google::protobuf::Message &from)
- void MergeFrom (const ::google::protobuf::Message &from)
- void CopyFrom (const TriangularCell &from)
- void MergeFrom (const TriangularCell &from)
- void Clear ()
- bool IsInitialized () const
- int ByteSize () const
- bool MergePartialFromCodedStream (::google::protobuf::io::CodedInputStream *input)
- void SerializeWithCachedSizes (::google::protobuf::io::CodedOutputStream *output) const
- ::google::protobuf::uint8 * SerializeWithCachedSizesToArray (::google::protobuf::uint8 *output) const
- int GetCachedSize () const
- ::google::protobuf::Metadata GetMetadata () const
- int coordinate_size () const
- void clear_coordinate ()
- const ::Configurator::Coordinate & coordinate (int index) const
- inline::Configurator::Coordinate * mutable_coordinate (int index)
- inline::Configurator::Coordinate * add_coordinate ()
- const ::google::protobuf::RepeatedPtrField< ::Configurator::Coordinate > & coordinate () const
- inline::google::protobuf::RepeatedPtrField< ::Configurator::Coordinate > * mutable_coordinate ()

7.71.2 Static Public Member Functions

- static const ::google::protobuf::Descriptor * descriptor ()
- static const TriangularCell & default_instance ()

7.71.3 Static Public Attributes

- static const int kCoordinateFieldNumber = 1

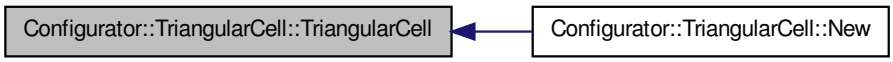
7.71.4 Friends

- void protobuf_AddDesc_confproblem_2eproto ()
- void protobuf_AssignDesc_confproblem_2eproto ()
- void protobuf_ShutdownFile_confproblem_2eproto ()

7.71.5 Constructor & Destructor Documentation

Configurator::TriangularCell::TriangularCell ( )

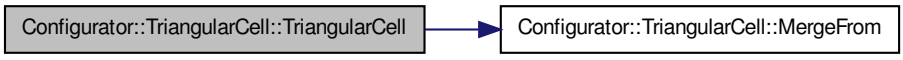
Here is the caller graph for this function:



Configurator::TriangularCell::~TriangularCell ( ) `[virtual]`

Configurator::TriangularCell::TriangularCell ([const TriangularCell &]from)

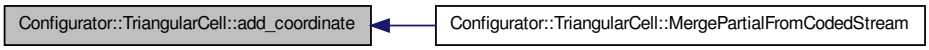
Here is the call graph for this function:



7.71.6 Member Function Documentation

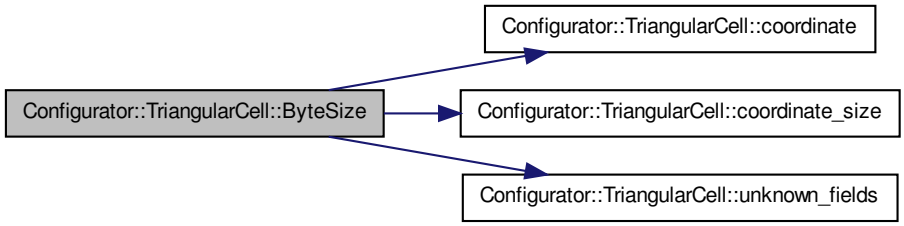
Configurator::Coordinate * Configurator::TriangularCell::add_coordinate ( ) `[inline]`

Here is the caller graph for this function:



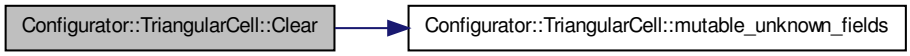
int Configurator::TriangularCell::ByteSize ( ) const

Here is the call graph for this function:

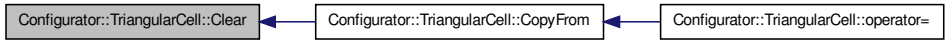


void Configurator::TriangularCell::Clear ( )

Here is the call graph for this function:



Here is the caller graph for this function:

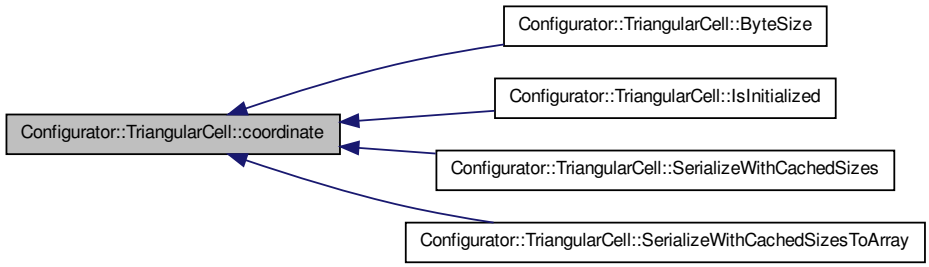


void Configurator::TriangularCell::clear_coordinate ( ) `[inline]`

const ::Configurator::Coordinate & Configurator::TriangularCell::coordinate ([int]index) const `[inline]`

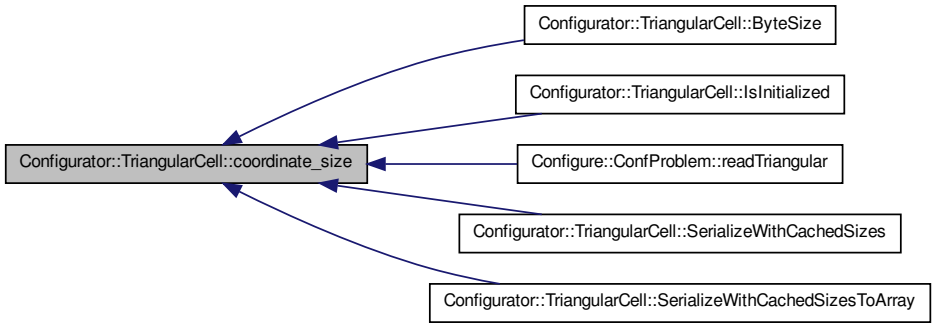
const ::google::protobuf::RepeatedPtrField<::Configurator::Coordinate > & Configurator::TriangularCell::coordinate ( ) const `[inline]`

Here is the caller graph for this function:



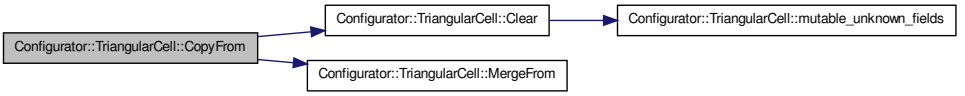
int Configurator::TriangularCell::coordinate_size ( ) const [inline]

Here is the caller graph for this function:



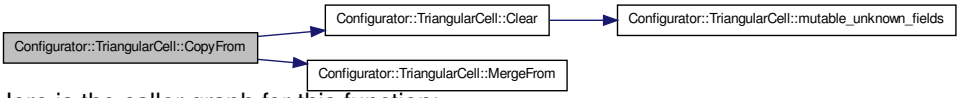
void Configurator::TriangularCell::CopyFrom ([const TriangularCell &]from)

Here is the call graph for this function:



void Configurator::TriangularCell::CopyFrom ([const ::google::protobuf::Message &]from)

Here is the call graph for this function:



Here is the caller graph for this function:

const TriangularCell & Configurator::TriangularCell::default_instance ( ) [static]

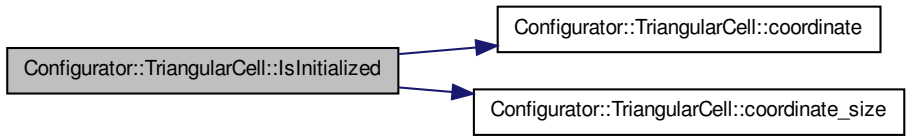
const ::google::protobuf::Descriptor * Configurator::TriangularCell::descriptor ( ) [static]

int Configurator::TriangularCell::GetCachedSize ( ) const [inline]

google::protobuf::Metadata Configurator::TriangularCell::GetMetadata ( ) const

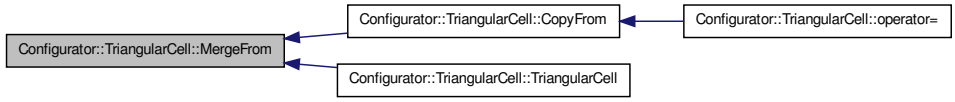
bool Configurator::TriangularCell::IsInitialized ( ) const

Here is the call graph for this function:



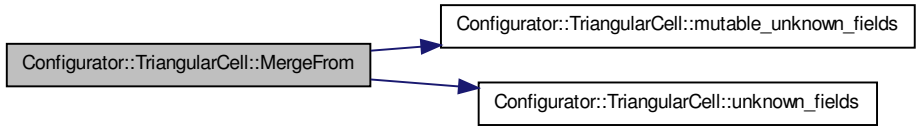
void Configurator::TriangularCell::MergeFrom ([const ::google::protobuf::Message &]from)

Here is the caller graph for this function:



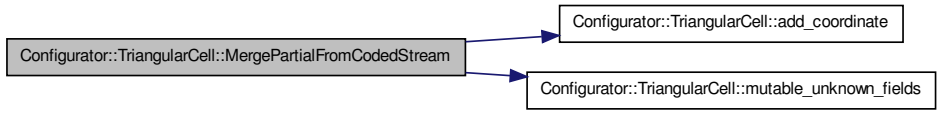
void Configurator::TriangularCell::MergeFrom ([const TriangularCell &]from)

Here is the call graph for this function:



bool Configurator::TriangularCell::MergePartialFromCodedStream ([::google::protobuf::io::CodedInputStream ∗]input)

Here is the call graph for this function:

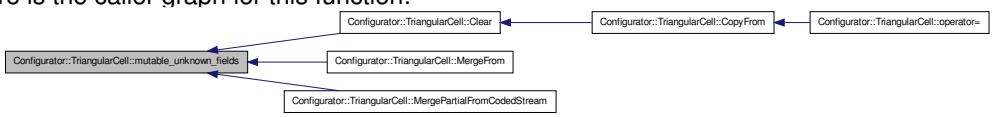


Configurator::Coordinate ∗ Configurator::TriangularCell::mutable_coordinate ([int]index) `[inline]`

google::protobuf::RepeatedPtrField<::Configurator::Coordinate > ∗ Configurator::TriangularCell::mutable_coordinate ( ) `[inline]`

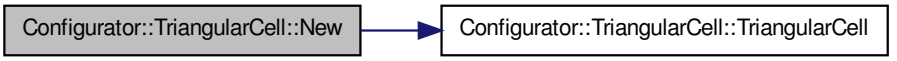
inline ::google::protobuf::UnknownFieldSet∗ Configurator::TriangularCell::mutable_unknown_fields ( ) `[inline]`

Here is the caller graph for this function:



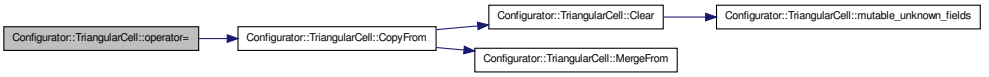
TriangularCell * Configurator::TriangularCell::New ( ) const

Here is the call graph for this function:



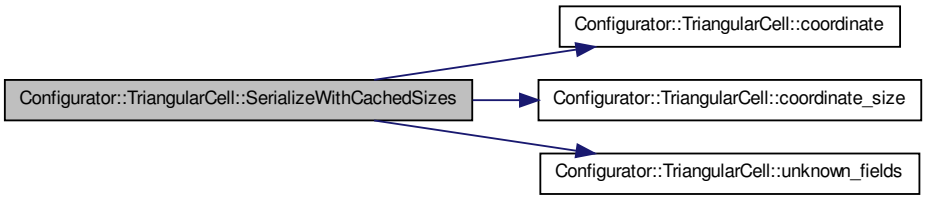
TriangularCell& Configurator::TriangularCell::operator= ([const TriangularCell &]from) [inline]

Here is the call graph for this function:



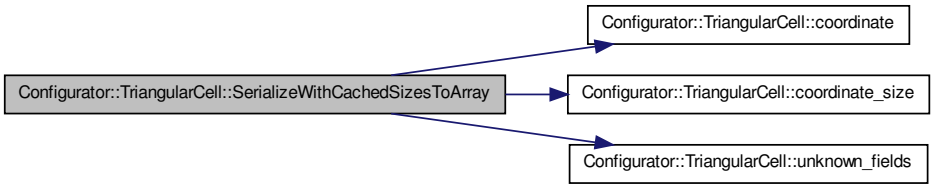
void Configurator::TriangularCell::SerializeWithCachedSizes ([::google::protobuf::io::CodedOutputStream *]output) const

Here is the call graph for this function:



google::protobuf::uint8 * Configurator::TriangularCell::SerializeWithCachedSizesToArray ([::google::protobuf::uint8 *]output) const

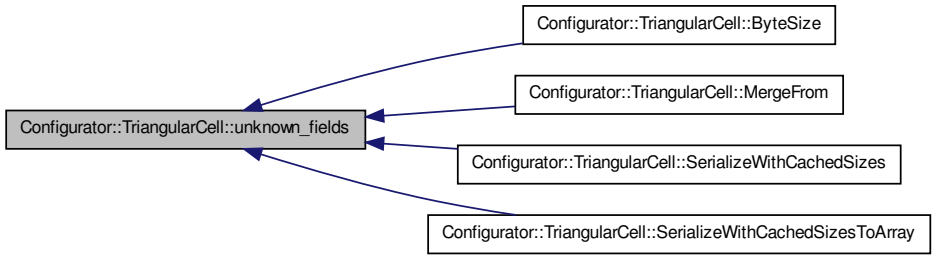
Here is the call graph for this function:



void Configurator::TriangularCell::Swap ([TriangularCell *]other)

const ::google::protobuf::UnknownFieldSet& Configurator::TriangularCell::unknown_-fields ( ) const [inline]

Here is the caller graph for this function:



7.71.7 Friends And Related Function Documentation

void protobuf_AddDesc_confproblem_2eproto ( ) [friend]

void protobuf_AssignDesc_confproblem_2eproto ( ) [friend]

void protobuf_ShutdownFile_confproblem_2eproto ( ) [friend]

7.71.8 Member Data Documentation

const int Configurator::TriangularCell::kCoordinateFieldNumber = 1 [static]

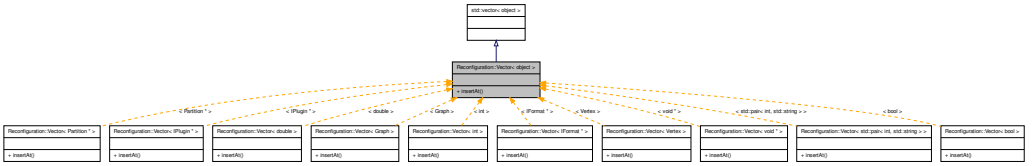
The documentation for this class was generated from the following files:

- confproblem.pb.h
- confproblem.pb.cc

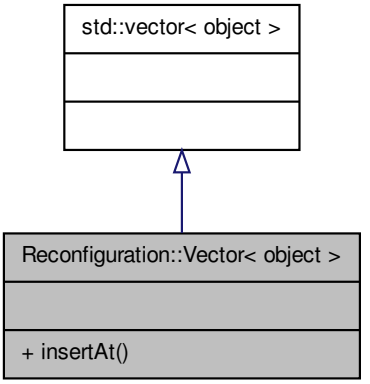
7.72 Reconfiguration::Vector< object > Class Template Reference

`#include <Vector.h>`

Inheritance diagram for Reconfiguration::Vector< object >:



Collaboration diagram for Reconfiguration::Vector< object >:



7.72.1 Public Member Functions

- void insertAt (unsigned int iPos, object oItem)

  Allows the insertion of an element at a certain position of a vector.

7.72.2 Detailed Description

template<class object> class Reconfiguration::Vector< object >

Extends the features of the standard vector class.

7.72.3 Member Function Documentation

template<class object> void Reconfiguration::Vector< object >::insertAt ([unsigned int]iPos, object *oItem*) `[inline]`

Allows the insertion of an element at a certain position of a vector.

Parameters

in	*iPos*	Vector position to be inserted.
in	*oItem*	Object to be inserted.

Returns

void

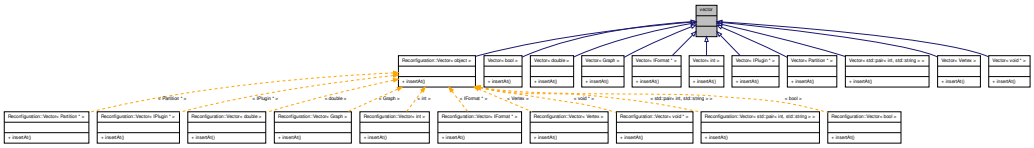
Inserts an object at a specific position of the vector. The rest of the elements go one position to the end. The length is automatically incremented.

The documentation for this class was generated from the following file:

- Vector.h

7.73 vector Class Reference

Inheritance diagram for vector:



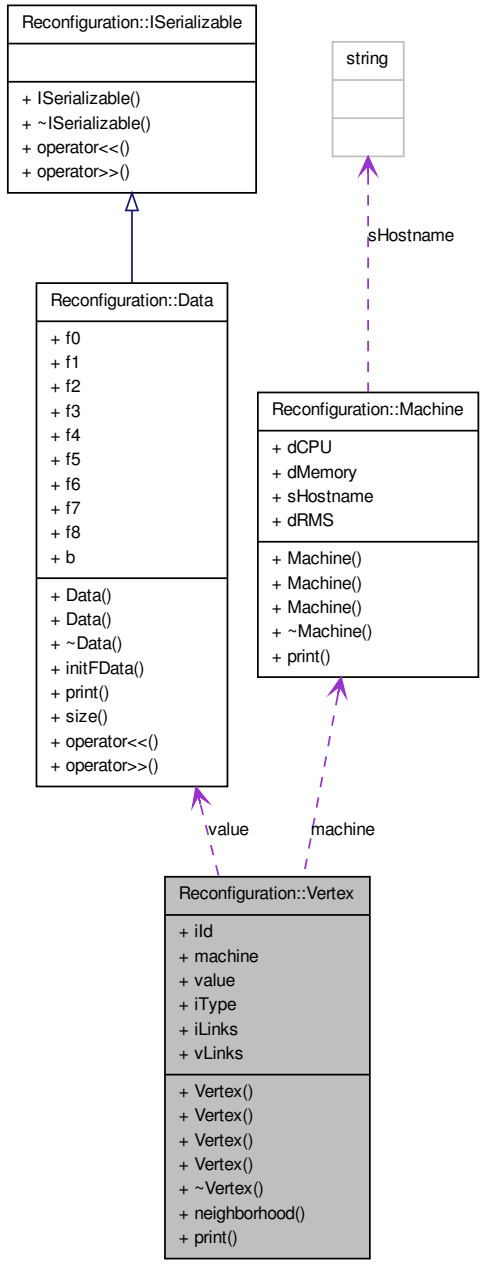
The documentation for this class was generated from the following file:

- Vector.h

7.74 Reconfiguration::Vertex Class Reference

`#include <Graph.h>`

Collaboration diagram for Reconfiguration::Vertex:



7.74.1 Public Member Functions

- Vertex ()

  Constructor of the Vertex class.

- Vertex (const Vertex &)

  Copy constructor of the Vertex class.

- Vertex (int, double, double, const std::string &)

  Constructor of the Vertex class.

- Vertex (int, const Data &)

  Constructor of the Vertex class.

- virtual ∼Vertex ()

  virtual destructor of the class Vertex.

- Vector< int > neighborhood ()

  Calculates the neighborhood of the actual vertex.

- void print ()

  Prints a Vertex object.

7.74.2 Public Attributes

- int iId
- Machine ∗ machine
- Data ∗ value
- int iType
- int iLinks
- std::list< Link > vLinks

7.74.3 Detailed Description

Graph vertex. Allocates information about a machine of the cluster or information about customized values of the problem.

7.74.4 Constructor & Destructor Documentation

Reconfiguration::Vertex::Vertex ( )

Constructor of the Vertex class.

Returns

∗this

Creates an object Vertex.

Reconfiguration::Vertex::Vertex ([const Vertex &]vertex)

Copy constructor of the Vertex class.

Parameters

in	*vertex*	Object to be copied.

Returns

∗this

Copies an object Vertex.

Reconfiguration::Vertex::Vertex ([int]iId, double *dCPU,* double *dMemory,* const std::string & *sName*)

Constructor of the Vertex class.

Parameters

in	*iId*	Vertex identifier.
in	*dCPU*	CPU frequency (MHz).
in	*dMemory*	Free available memory (Mb).
in	*sName*	Machine hostname.

Returns

∗this

Creates an object Vertex setting its internal variables, when the vertex is a machine object.

See also

enum VERTEXTYPE.

Parameters

in	*iId*	Vertex identifier.
in	*dCPU*	CPU frequency (MHz).
in	*dMemory*	Free available memory (Mb).
in	*sName*	Machine hostname.

Returns

∗this

Creates an object Vertex setting its internal variables, when the vertex is a machine object.

See also

enum VERTEXTYPE

Reconfiguration::Vertex::Vertex ([int]iId, const Data & *value*)

Constructor of the Vertex class.

Parameters

in	*iId*	Vertex identifier.
in	*data*	Application graph cell information.

Returns

∗this

Creates an object Vertex setting its internal variables, when the vertex is a data object.

See also

enum VERTEXTYPE

Parameters

in	*iId*	Vertex identifier.
in	*Value*	Application graph cell information.

Returns

∗this

Creates an object Vertex setting its internal variables, when the vertex is a data object.

See also

enum VERTEXTYPE

Reconfiguration::Vertex::∼Vertex ( ) `[virtual]`

virtual destructor of the class Vertex.

Returns

void

Destroys the Vertex object by freeing the links vector.

7.74.5 Member Function Documentation

Vector< int > Reconfiguration::Vertex::neighborhood ( )

Calculates the neighborhood of the actual vertex.

Returns

Vector<int>

Calculates the neighborhood of the actual vertex visiting its vector of links and taking the index of each one.

void Reconfiguration::Vertex::print ( )

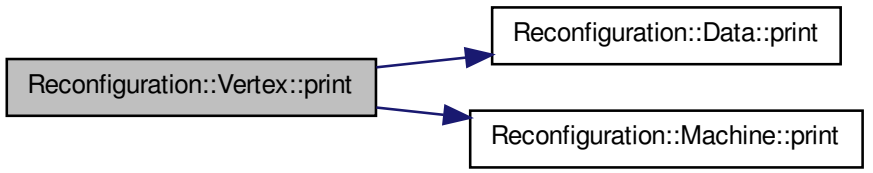
Prints a Vertex object.

Returns

void

Prints the properties of the Vertex object.

Here is the call graph for this function:



7.74.6 Member Data Documentation

int Reconfiguration::Vertex::iId

Identifier of the vertex.

int Reconfiguration::Vertex::iLinks

Number of links of the actual vertex.

int Reconfiguration::Vertex::iType

Identifies if the vertex is a machine or a data vertex.

See also

enum VERTEXTYPE.

Machine Reconfiguration::Vertex::machine

In case the graph is a system graph, the vertex is a machine of the cluster.

Data Reconfiguration::Vertex::value

In case the graph is an application graph, the vertex is a cell of the problem.

std::list< Link > Reconfiguration::Vertex::vLinks

List of link objects of the actual vertex.

The documentation for this class was generated from the following files:

- Graph.h
- Graph.cpp

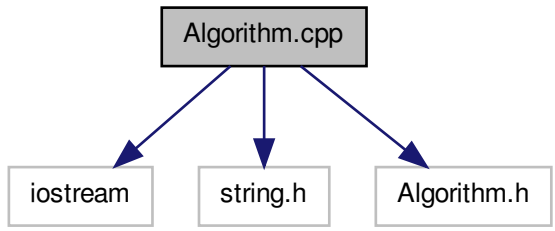
8 File Documentation

8.1 Algorithm.cpp File Reference

Abstract class for implementing the algorithm. This layer is needed to pass the user Algorithm variables to the kernel of the middleware.

```
#include <iostream>

#include <string.h>

#include "Algorithm.h"
```

Include dependency graph for Algorithm.cpp:



8.1.1 Namespaces

- namespace Reconfiguration

8.1.2 Detailed Description

Abstract class for implementing the algorithm. This layer is needed to pass the user Algorithm variables to the kernel of the middleware.

Author

Carmen B. Navarrete

Date

16-Jul-2009

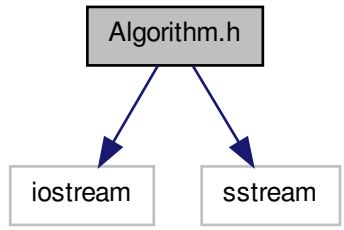
04-Feb-2011

8.2 Algorithm.h File Reference

Header file for Algorithm.cpp file.

```
#include <iostream>

#include <sstream>
```

Include dependency graph for Algorithm.h:



8.2.1 Classes

- class Reconfiguration::Algorithm

8.2.2 Namespaces

- namespace Reconfiguration

8.2.3 Detailed Description

Header file for Algorithm.cpp file.

Author

Carmen B. Navarrete

Date

16-Jul-2009

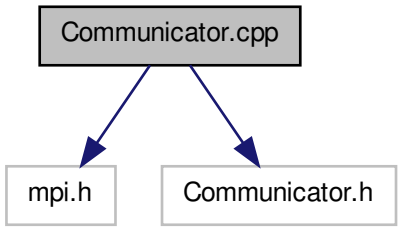
04-Feb-2011

8.3 Communicator.cpp File Reference

Improves the functionality of the MPI layer by adding other functions needed for the framework.

```
#include <mpi.h>

#include "Communicator.h"
```

Include dependency graph for Communicator.cpp:



8.3.1 Namespaces

- namespace Reconfiguration

8.3.2 Detailed Description

Improves the functionality of the MPI layer by adding other functions needed for the framework.

Author

Carmen B. Navarrete

Date

14-ago-2009

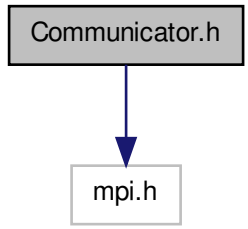
07-Feb-2011

8.4 Communicator.h File Reference

Header file for Communicator.cpp file.

```
#include <mpi.h>
```

Include dependency graph for Communicator.h:



8.4.1 Classes

- class Reconfiguration::Communicator

8.4.2 Namespaces

- namespace Reconfiguration

8.4.3 Defines

- #define TAG_INVALID -1

  Invalid tag identifier.

8.4.4 Detailed Description

Header file for Communicator.cpp file.

Author

Carmen B. Navarrete

Date

14-Ago-2009

07-Feb-2011

8.4.5 Define Documentation

#define TAG_INVALID -1

Invalid tag identifier.

8.5 confcluster.pb.cc File Reference

```
#include "confcluster.pb.h"

   #include <google/protobuf/stubs/once.h>

   #include <google/protobuf/io/coded_stream.h>

   #include <google/protobuf/wire_format_lite_inl.h>

   #include <google/protobuf/descriptor.h>

   #include <google/protobuf/reflection_ops.h>

   #include <google/protobuf/wire_format.h>
```

Include dependency graph for confcluster.pb.cc:

8.5.1 Classes

- struct Configurator::StaticDescriptorInitializer_confcluster_2eproto

8.5.2 Namespaces

- namespace Configurator
- namespace Configurator::cc

8.5.3 Defines

- #define INTERNAL_SUPPRESS_PROTOBUF_FIELD_DEPRECATION
- #define DO_(EXPRESSION) if (!(EXPRESSION)) return false
- #define DO_(EXPRESSION) if (!(EXPRESSION)) return false
- #define DO_(EXPRESSION) if (!(EXPRESSION)) return false

8.5.4 Functions

- void Configurator::protobuf_AssignDesc_confcluster_2eproto ()

- void Configurator::protobuf_ShutdownFile_confcluster_2eproto ()
- void Configurator::protobuf_AddDesc_confcluster_2eproto ()

8.5.5 Variables

- const ::google::protobuf::Descriptor * Configurator::cc::Node_descriptor_ = NULL
- const ::google::protobuf::internal::GeneratedMessageReflection * Configurator::cc::Node_-reflection_ = NULL
- const ::google::protobuf::Descriptor * Configurator::cc::Link_descriptor_ = NULL
- const ::google::protobuf::internal::GeneratedMessageReflection * Configurator::cc::Link_-reflection_ = NULL
- const ::google::protobuf::Descriptor * Configurator::cc::Cluster_descriptor_ = NULL
- const ::google::protobuf::internal::GeneratedMessageReflection * Configurator::cc::Cluster_-reflection_ = NULL
- struct Configurator::StaticDescriptorInitializer_confcluster_2eproto Configurator::static_-descriptor_initializer_confcluster_2eproto_

8.5.6 Define Documentation

#define DO_([]EXPRESSION) if (!(EXPRESSION)) return false

#define DO_([]EXPRESSION) if (!(EXPRESSION)) return false

#define DO_([]EXPRESSION) if (!(EXPRESSION)) return false

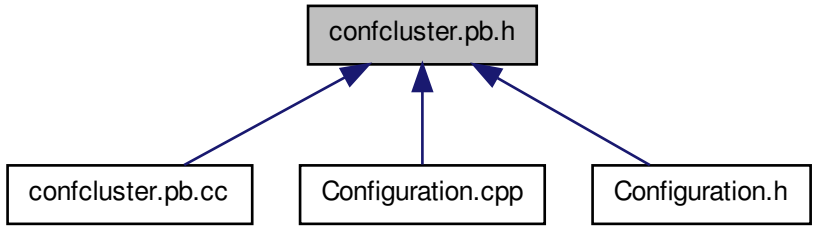
#define INTERNAL_SUPPRESS_PROTOBUF_FIELD_DEPRECATION

8.6 confcluster.pb.h File Reference

```
#include <string>

   #include <google/protobuf/stubs/common.h>

   #include <google/protobuf/generated_message_util.h>

   #include <google/protobuf/repeated_field.h>

   #include <google/protobuf/extension_set.h>

   #include <google/protobuf/generated_message_reflection.h>
```

Include dependency graph for confcluster.pb.h:

This graph shows which files directly or indirectly include this file:



8.6.1 Classes

- class Configurator::Node
- class Configurator::Link
- class Configurator::Cluster

8.6.2 Namespaces

- namespace Configurator
- namespace google
- namespace google::protobuf

8.6.3 Functions

- void Configurator::protobuf_AddDesc_confcluster_2eproto ()
- void Configurator::protobuf_AssignDesc_confcluster_2eproto ()
- void Configurator::protobuf_ShutdownFile_confcluster_2eproto ()

8.7 Configuration.cpp File Reference

API for managing the Configuration class.

```
#include <algorithm>

#include <fstream>

#include <fcntl.h>

#include <google/protobuf/text_format.h>

#include <google/protobuf/io/zero_copy_stream.h>

#include <google/protobuf/io/zero_copy_stream_impl.h>

#include <iostream>

#include <sstream>

#include <string>

#include "Defines.h"

#include "confcluster.pb.h"

#include "confloadbalancer.pb.h"

#include "confpartitioner.pb.h"

#include "confproblem.pb.h"

#include "Configuration.h"

#include "Coordinate.h"

#include "Exception.h"

#include "String.h"
```

Include dependency graph for Configuration.cpp:

8.7.1 Namespaces

- namespace Configure

8.7.2 Detailed Description

API for managing the Configuration class.

Author

Carmen B. Navarrete

Date

06-Jun-2009

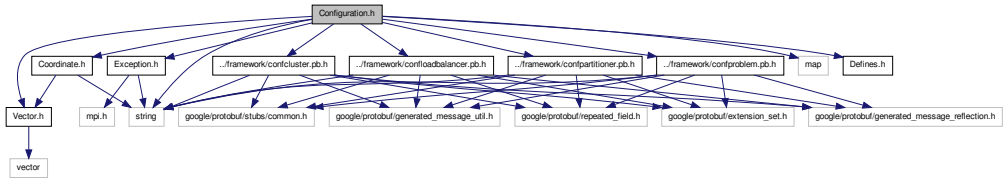
21-Feb-2011

8.8 Configuration.h File Reference

Header file for Configuration.cpp file.

```
#include <string>

#include <map>

#include "Coordinate.h"

#include "Defines.h"

#include "Exception.h"

#include "../framework/confcluster.pb.h"

#include "../framework/confloadbalancer.pb.h"

#include "../framework/confpartitioner.pb.h"

#include "../framework/confproblem.pb.h"

#include "Vector.h"
```

Include dependency graph for Configuration.h:



8.8.1 Classes

- class Configure::Configuration
- class Configure::ConfCluster
- class Configure::ConfLoadBalancer
- class Configure::ConfPartitioner
- class Configure::ConfProblem

8.8.2 Namespaces

- namespace Configure

8.8.3 Detailed Description

Header file for Configuration.cpp file.

Author

Carmen B. Navarrete

Date

06-Jun-2009

21-Feb-2011

8.9 confloadbalancer.pb.cc File Reference

```
#include "confloadbalancer.pb.h"

   #include <google/protobuf/stubs/once.h>

   #include <google/protobuf/io/coded_stream.h>

   #include <google/protobuf/wire_format_lite_inl.h>

   #include <google/protobuf/descriptor.h>

   #include <google/protobuf/reflection_ops.h>

   #include <google/protobuf/wire_format.h>
```

Include dependency graph for confloadbalancer.pb.cc:

8.9.1 Classes

- struct Configurator::StaticDescriptorInitializer_confloadbalancer_2eproto

8.9.2 Namespaces

- namespace Configurator

8.9.3 Defines

- #define INTERNAL_SUPPRESS_PROTOBUF_FIELD_DEPRECATION
- #define DO_(EXPRESSION) if (!(EXPRESSION)) return false
- #define DO_(EXPRESSION) if (!(EXPRESSION)) return false
- #define DO_(EXPRESSION) if (!(EXPRESSION)) return false

8.9.4 Functions

- void Configurator::protobuf_AssignDesc_confloadbalancer_2eproto ()
- void Configurator::protobuf_ShutdownFile_confloadbalancer_2eproto ()
- void Configurator::protobuf_AddDesc_confloadbalancer_2eproto ()

8.9.5 Variables

- struct Configurator::StaticDescriptorInitializer_confloadbalancer_2eproto Configurator::static_-descriptor_initializer_confloadbalancer_2eproto_

8.9.6 Define Documentation

#define DO_([]EXPRESSION) if (!(EXPRESSION)) return false

#define DO_([]EXPRESSION) if (!(EXPRESSION)) return false

#define DO_([]EXPRESSION) if (!(EXPRESSION)) return false

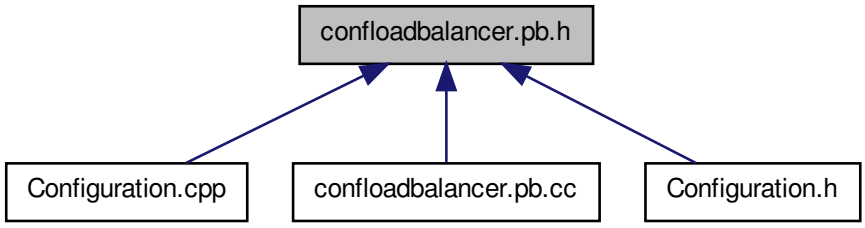
#define INTERNAL_SUPPRESS_PROTOBUF_FIELD_DEPRECATION

8.10 confloadbalancer.pb.h File Reference

```
#include <string>

   #include <google/protobuf/stubs/common.h>

   #include <google/protobuf/generated_message_util.h>

   #include <google/protobuf/repeated_field.h>

   #include <google/protobuf/extension_set.h>

   #include <google/protobuf/generated_message_reflection.h>
```

Include dependency graph for confloadbalancer.pb.h:

This graph shows which files directly or indirectly include this file:



8.10.1 Classes

- class Configurator::Plugin
- class Configurator::Historical
- class Configurator::LoadBalancer

8.10.2 Namespaces

- namespace Configurator
- namespace google
- namespace google::protobuf

8.10.3 Functions

- void Configurator::protobuf_AddDesc_confloadbalancer_2eproto ()
- void Configurator::protobuf_AssignDesc_confloadbalancer_2eproto ()
- void Configurator::protobuf_ShutdownFile_confloadbalancer_2eproto ()

8.11 confpartitioner.pb.cc File Reference

```
#include "confpartitioner.pb.h"

   #include <google/protobuf/stubs/once.h>

   #include <google/protobuf/io/coded_stream.h>

   #include <google/protobuf/wire_format_lite_inl.h>

   #include <google/protobuf/descriptor.h>

   #include <google/protobuf/reflection_ops.h>

   #include <google/protobuf/wire_format.h>
```

Include dependency graph for confpartitioner.pb.cc:

8.11.1 Classes

- struct Configurator::StaticDescriptorInitializer_confpartitioner_2eproto

8.11.2 Namespaces

- namespace Configurator

8.11.3 Defines

- #define INTERNAL_SUPPRESS_PROTOBUF_FIELD_DEPRECATION
- #define DO_(EXPRESSION) if (!(EXPRESSION)) return false
- #define DO_(EXPRESSION) if (!(EXPRESSION)) return false
- #define DO_(EXPRESSION) if (!(EXPRESSION)) return false
- #define DO_(EXPRESSION) if (!(EXPRESSION)) return false
- #define DO_(EXPRESSION) if (!(EXPRESSION)) return false
- #define DO_(EXPRESSION) if (!(EXPRESSION)) return false
- #define DO_(EXPRESSION) if (!(EXPRESSION)) return false
- #define DO_(EXPRESSION) if (!(EXPRESSION)) return false

8.11.4 Functions

- void Configurator::protobuf_AssignDesc_confpartitioner_2eproto ()
- void Configurator::protobuf_ShutdownFile_confpartitioner_2eproto ()
- void Configurator::protobuf_AddDesc_confpartitioner_2eproto ()
- const ::google::protobuf::EnumDescriptor * Configurator::Method_Type_descriptor ()
- bool Configurator::Method_Type_IsValid (int value)

8.11.5 Variables

- struct Configurator::StaticDescriptorInitializer_confpartitioner_2eproto Configurator::static_descriptor_initializer_confpartitioner_2eproto_

8.11.6 Define Documentation

#define DO_([]EXPRESSION) if (!(EXPRESSION)) return false

#define DO_([]EXPRESSION) if (!(EXPRESSION)) return false

#define DO_([]EXPRESSION) if (!(EXPRESSION)) return false

#define DO_([]EXPRESSION) if (!(EXPRESSION)) return false

#define DO_([]EXPRESSION) if (!(EXPRESSION)) return false

#define DO_([]EXPRESSION) if (!(EXPRESSION)) return false

#define DO_([]EXPRESSION) if (!(EXPRESSION)) return false

#define DO_([]EXPRESSION) if (!(EXPRESSION)) return false

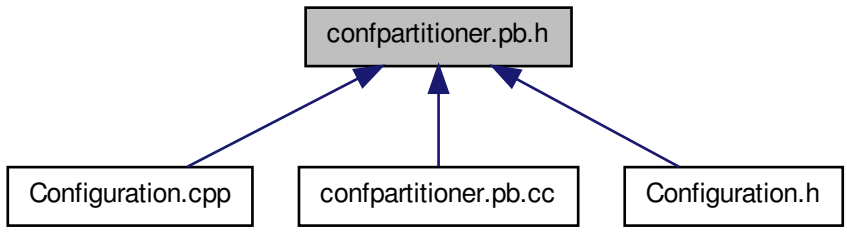
#define INTERNAL_SUPPRESS_PROTOBUF_FIELD_DEPRECATION

8.12 confpartitioner.pb.h File Reference

```
#include <string>

   #include <google/protobuf/stubs/common.h>

   #include <google/protobuf/generated_message_util.h>

   #include <google/protobuf/repeated_field.h>

   #include <google/protobuf/extension_set.h>

   #include <google/protobuf/generated_message_reflection.h>
```

Include dependency graph for confpartitioner.pb.h:

This graph shows which files directly or indirectly include this file:



8.12.1 Classes

- class Configurator::Constant
- class Configurator::Lineal
- class Configurator::Annealing
- class Configurator::PID
- class Configurator::Kalman
- class Configurator::Heuristic
- class Configurator::Method
- class Configurator::Partitioner

8.12.2 Namespaces

- namespace Configurator
- namespace google
- namespace google::protobuf

8.12.3 Enumerations

- enum Configurator::Method_Type { Configurator::Method_Type_STATIC = 1, Configurator::Method_Type_DYNAMIC = 2 }

8.12.4 Functions

- void Configurator::protobuf_AddDesc_confpartitioner_2eproto ()
- void Configurator::protobuf_AssignDesc_confpartitioner_2eproto ()
- void Configurator::protobuf_ShutdownFile_confpartitioner_2eproto ()
- bool Configurator::Method_Type_IsValid (int value)
- const ::google::protobuf::EnumDescriptor * Configurator::Method_Type_descriptor ()
- const ::std::string & Configurator::Method_Type_Name (Method_Type value)
- bool Configurator::Method_Type_Parse (const ::std::string &name, Method_Type *value)
- template<>
  const EnumDescriptor * google::protobuf::GetEnumDescriptor< ::Configurator::Method_Type > ()

8.12.5 Variables

- const Method_Type Configurator::Method_Type_Type_MIN = Method_Type_STATIC
- const Method_Type Configurator::Method_Type_Type_MAX = Method_Type_DYNAMIC
- const int Configurator::Method_Type_Type_ARRAYSIZE = Method_Type_Type_MAX + 1

8.13 confproblem.pb.cc File Reference

```
#include "confproblem.pb.h"

   #include <google/protobuf/stubs/once.h>

   #include <google/protobuf/io/coded_stream.h>

   #include <google/protobuf/wire_format_lite_inl.h>

   #include <google/protobuf/descriptor.h>

   #include <google/protobuf/reflection_ops.h>

   #include <google/protobuf/wire_format.h>
```

Include dependency graph for confproblem.pb.cc:

8.13.1 Classes

- struct Configurator::StaticDescriptorInitializer_confproblem_2eproto

8.13.2 Namespaces

- namespace Configurator

8.13.3 Defines

- #define INTERNAL_SUPPRESS_PROTOBUF_FIELD_DEPRECATION
- #define DO_(EXPRESSION) if (!(EXPRESSION)) return false
- #define DO_(EXPRESSION) if (!(EXPRESSION)) return false
- #define DO_(EXPRESSION) if (!(EXPRESSION)) return false
- #define DO_(EXPRESSION) if (!(EXPRESSION)) return false
- #define DO_(EXPRESSION) if (!(EXPRESSION)) return false
- #define DO_(EXPRESSION) if (!(EXPRESSION)) return false
- #define DO_(EXPRESSION) if (!(EXPRESSION)) return false
- #define DO_(EXPRESSION) if (!(EXPRESSION)) return false

- #define DO_(EXPRESSION) if (!(EXPRESSION)) return false
- #define DO_(EXPRESSION) if (!(EXPRESSION)) return false
- #define DO_(EXPRESSION) if (!(EXPRESSION)) return false
- #define DO_(EXPRESSION) if (!(EXPRESSION)) return false
- #define DO_(EXPRESSION) if (!(EXPRESSION)) return false

8.13.4 Functions

- void Configurator::protobuf_AssignDesc_confproblem_2eproto ()
- void Configurator::protobuf_ShutdownFile_confproblem_2eproto ()
- void Configurator::protobuf_AddDesc_confproblem_2eproto ()
- const ::google::protobuf::EnumDescriptor * Configurator::Output_How_descriptor ()
- bool Configurator::Output_How_IsValid (int value)

8.13.5 Variables

- struct Configurator::StaticDescriptorInitializer_confproblem_2eproto Configurator::static_-descriptor_initializer_confproblem_2eproto_

8.13.6 Define Documentation

#define DO_([]EXPRESSION) if (!(EXPRESSION)) return false

#define DO_([]EXPRESSION) if (!(EXPRESSION)) return false

#define DO_([]EXPRESSION) if (!(EXPRESSION)) return false

#define DO_([]EXPRESSION) if (!(EXPRESSION)) return false

#define DO_([]EXPRESSION) if (!(EXPRESSION)) return false

#define DO_([]EXPRESSION) if (!(EXPRESSION)) return false

#define DO_([]EXPRESSION) if (!(EXPRESSION)) return false

#define DO_([]EXPRESSION) if (!(EXPRESSION)) return false

#define DO_([]EXPRESSION) if (!(EXPRESSION)) return false

#define DO_([]EXPRESSION) if (!(EXPRESSION)) return false

#define DO_([]EXPRESSION) if (!(EXPRESSION)) return false

#define DO_([]EXPRESSION) if (!(EXPRESSION)) return false

#define DO_([]EXPRESSION) if (!(EXPRESSION)) return false

#define INTERNAL_SUPPRESS_PROTOBUF_FIELD_DEPRECATION

8.14 confproblem.pb.h File Reference

```
#include <string>
```

```
#include <google/protobuf/stubs/common.h>
```

```
#include <google/protobuf/generated_message_util.h>
```

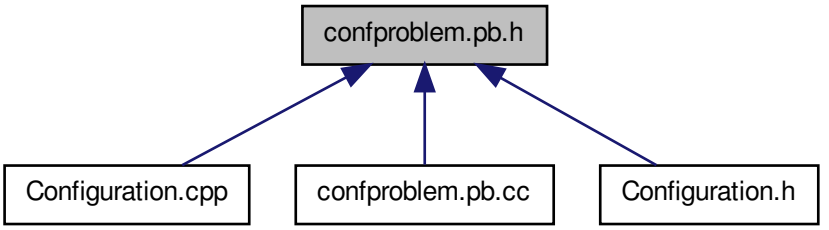
```
#include <google/protobuf/repeated_field.h>
```

```
#include <google/protobuf/extension_set.h>
```

```
#include <google/protobuf/generated_message_reflection.h>
```

Include dependency graph for confproblem.pb.h:

This graph shows which files directly or indirectly include this file:



8.14.1 Classes

- class Configurator::Coordinate
- class Configurator::SquareCell
- class Configurator::TriangularCell
- class Configurator::HexagonalCell
- class Configurator::Cell
- class Configurator::Connection
- class Configurator::Matrix
- class Configurator::Graph
- class Configurator::Topology

- class Configurator::OutputFormat
- class Configurator::Input
- class Configurator::Output
- class Configurator::Simulation

8.14.2 Namespaces

- namespace Configurator
- namespace google
- namespace google::protobuf

8.14.3 Enumerations

- enum Configurator::Output_How { Configurator::Output_How_GLOBAL = 1, Configurator::Output_How_LOCAL = 2 }

8.14.4 Functions

- void Configurator::protobuf_AddDesc_confproblem_2eproto ()
- void Configurator::protobuf_AssignDesc_confproblem_2eproto ()
- void Configurator::protobuf_ShutdownFile_confproblem_2eproto ()
- bool Configurator::Output_How_IsValid (int value)
- const ::google::protobuf::EnumDescriptor * Configurator::Output_How_descriptor ()
- const ::std::string & Configurator::Output_How_Name (Output_How value)
- bool Configurator::Output_How_Parse (const ::std::string &name, Output_How *value)
- template<>
  const EnumDescriptor * google::protobuf::GetEnumDescriptor< ::Configurator::Output_How > ()

8.14.5 Variables

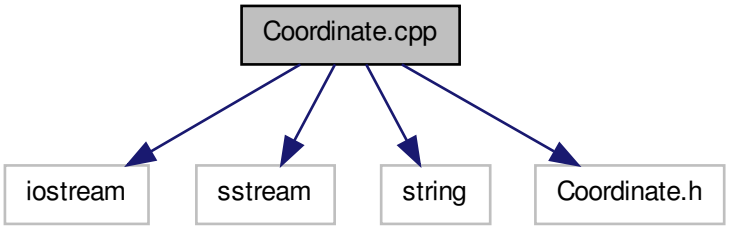
- const Output_How Configurator::Output_How_How_MIN = Output_How_GLOBAL
- const Output_How Configurator::Output_How_How_MAX = Output_How_LOCAL
- const int Configurator::Output_How_How_ARRAYSIZE = Output_How_How_MAX + 1

8.15 Coordinate.cpp File Reference

Set of ints to define the direction of the links at the configuration file $problem.conf written by the user.

```
#include <iostream>

#include <sstream>

#include <string>

#include "Coordinate.h"
```

Include dependency graph for Coordinate.cpp:



8.15.1 Namespaces

- namespace Reconfiguration

8.15.2 Detailed Description

Set of ints to define the direction of the links at the configuration file $problem.conf written by the user.

Author

Carmen B. Navarrete

Date

18-Ago-2009

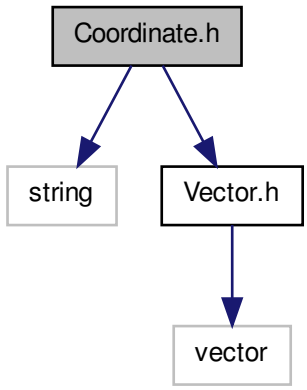
07-Feb-2011

8.16 Coordinate.h File Reference

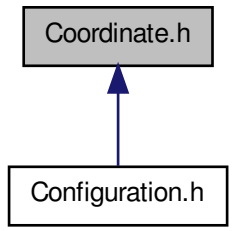
Header file for Coordinate.cpp file.

```
#include <string>

#include "Vector.h"
```

Include dependency graph for Coordinate.h:



This graph shows which files directly or indirectly include this file:



8.16.1 Classes

- class Reconfiguration::Coordinate

8.16.2 Namespaces

- namespace Reconfiguration

8.16.3 Detailed Description

Header file for Coordinate.cpp file.

Author

Carmen Navarrete

Date

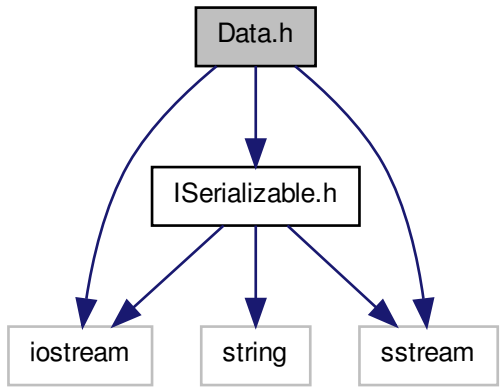
18-Ago-2009

07-Feb-2011

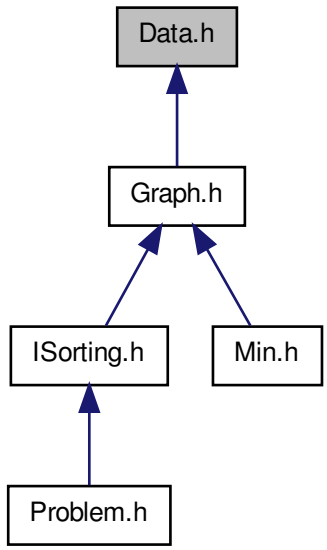
8.17 Data.h File Reference

```
#include "ISerializable.h"

   #include <iostream>

   #include <sstream>
```

Include dependency graph for Data.h:



This graph shows which files directly or indirectly include this file:



8.17.1 Classes

- class Reconfiguration::Data

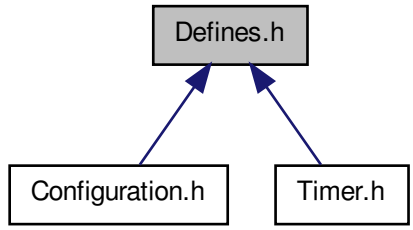
8.17.2 Namespaces

- namespace Reconfiguration

8.18 Defines.h File Reference

Common defines for the platform.

This graph shows which files directly or indirectly include this file:



8.18.1 Defines

- #define D1 1

  One dimension definition.

- #define PARAMETERS 5

  Number of parameters to run the platform (executable, cluster.conf, loadbalancer.conf, partitioner.conf, problem.conf).

- #define SUPERMASTER 0

  MPI rank for the global master of the platform.

- #define MAX_STRING 255

  Maximum length for a string.

- #define INVALID_TIMESTAMP -1

  Invalid timestamp for the timer.

- #define INITIAL_TIMESTAMP 0

  Starting point for the timer.

- #define PATH_RESULTS "../rst/"

  Path for saving the result files and logs.

8.18.2 Enumerations

- enum VARIANTS { STATIC, DYNAMIC }

  Partitioning variants: STATIC (no changes during execution) or DYNAMIC (posible changes during execution).

- enum WHICH { GLOBAL, LOCAL }

  Which partitions do we have to save: GLOBAL (all partitions in the same file) or LOCAL (each partition in an only file).

- enum VERTEXTYPE { MACHINEV, DATAV }

  Type of the vertex of the graph: MACHINEV (cluster machine vertex) or DATAV (data problem vertex).

- enum TAGS {
  TAG_JOIN, TAG_CONFIGURE, TAG_PARTITION, TAG_DATA,
  TAG_RESULTS, TAG_FINALIZE, TAG_END }

  Tags used by the system:

  TAG_JOIN: (S->M) Notify a new slave to the master.

  TAG_CONFIGURE: (M->S) Configuration parameters.

  TAG_PARTITION: (M->S) Partition information.

  TAG_DATA: (M->S) Data information.

  TAG_RESULTS: (S->M) Step results.

  TAG_FINALIZE: (M->S) Finalize the execution.

  TAG_END: (S->M) ACK of reception of TAG_FINALIZE.

- enum UPDATE { RO, RW }

  Vertices updating possibilities: RO (Read Only) or RW (Read-Write).

- enum DATAFRAME { PARTITION, DATA }

  Kind of dataframes.

8.18.3 Detailed Description

Common defines for the platform.

Author

Carmen B. Navarrete

Date

17-Jul-2009

08-Feb-2011

8.18.4 Define Documentation

#define D1 1

One dimension definition.

INITIAL_TIMESTAMP 0

Starting point for the timer.

INVALID_TIMESTAMP -1

Invalid timestamp for the timer.

MAX_STRING 255

Maximum length for a string.

#define PARAMETERS 5

Number of parameters to run the platform (executable, cluster.conf, loadbalancer.conf, partitioner.conf, problem.conf).

PATH_RESULTS "../rst/"

Path for saving the result files and logs.

#define SUPERMASTER 0

MPI rank for the global master of the platform.

8.18.5 Enumeration Type Documentation

enum DATAFRAME

Kind of dataframes.

Enumerator:

PARTITION

DATA

enum TAGS

Tags used by the system:

TAG_JOIN: (S->M) Notify a new slave to the master.

TAG_CONFIGURE: (M->S) Configuration parameters.

TAG_PARTITION: (M->S) Partition information.

TAG_DATA: (M->S) Data information.

TAG_RESULTS: (S->M) Step results.

TAG_FINALIZE: (M->S) Finalize the execution.

TAG_END: (S->M) ACK of reception of TAG_FINALIZE.

Enumerator:

TAG_JOIN

TAG_CONFIGURE

TAG_PARTITION

TAG_DATA

TAG_RESULTS

TAG_FINALIZE

TAG_END

enum UPDATE

Vertices updating possibilities: RO (Read Only) or RW (Read-Write).

Enumerator:

RO

RW

enum VARIANTS

Partitioning variants: STATIC (no changes during execution) or DYNAMIC (posible changes during execution).

Enumerator:

STATIC

DYNAMIC

enum VERTEXTYPE

Type of the vertex of the graph: MACHINEV (cluster machine vertex) or DATAV (data problem vertex).

Enumerator:

MACHINEV

DATAV

enum WHICH

Which partitions do we have to save: GLOBAL (all partitions in the same file) or LOCAL (each partition in an only file).

Enumerator:

GLOBAL

LOCAL

8.19 Errors.h File Reference

Common defines for the error management.

8.19.1 Defines

- #define OK 0

  Definition of OK.

- #define INVALID_ARGUMENTS -1

  Definition of INVALID_ARGUMENTS for the return error code of function Algorithm::Process.

- #define NO_SUCH_FILE -2

  Definition of NO_SUCH_FILE for the return error code of function Algorithm::Process.

- #define INVALID_SIZE -3

  Definition of INVALID_SIZE for the return error code of function Algorithm::Process.

8.19.2 Detailed Description

Common defines for the error management.

Author

Carmen B. Navarrete

Date

17-Jun-2010

08-Feb-2011

8.19.3 Define Documentation

INVALID_ARGUMENTS -1

Definition of INVALID_ARGUMENTS for the return error code of function Algorithm::Process.

INVALID_SIZE -3

Definition of INVALID_SIZE for the return error code of function Algorithm::Process.

NO_SUCH_FILE -2

Definition of NO_SUCH_FILE for the return error code of function Algorithm::Process.

OK 0

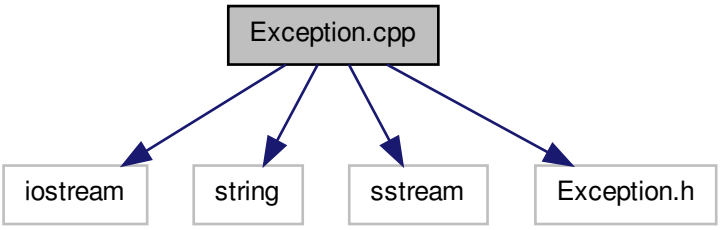
Definition of OK.

8.20 Exception.cpp File Reference

API for the customized exceptions that can be thrown during the execution of the framework.

```
#include <iostream>

#include <string>

#include <sstream>

#include "Exception.h"
```

Include dependency graph for Exception.cpp:



8.20.1 Namespaces

- namespace Reconfiguration

8.20.2 Detailed Description

API for the customized exceptions that can be thrown during the execution of the framework.

Author

Carmen B. Navarrete

Date

01-Abr-2009

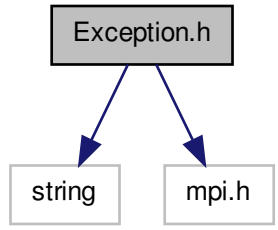
08-Feb-2011

8.21 Exception.h File Reference

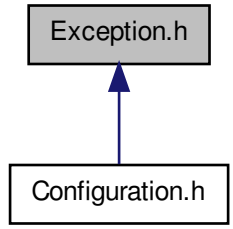
```
#include <string>
```

```
#include <mpi.h>
```

Include dependency graph for Exception.h:



This graph shows which files directly or indirectly include this file:



8.21.1 Classes

- class Reconfiguration::baseException
- class Reconfiguration::OutOfBoundsException
- class Reconfiguration::InvalidArgumentsException
- class Reconfiguration::NoSuchFileException
- class Reconfiguration::ConfigurationException
- class Reconfiguration::InternalException
- class Reconfiguration::MPIException

8.21.2 Namespaces

- namespace Reconfiguration

8.21.3 Detailed Description

Header file for Exception.cpp file.

Author

Carmen B. Navarrete

Date

01-Abr-2009

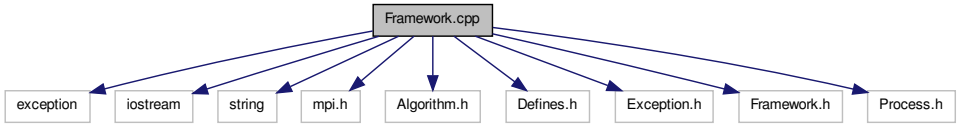
08-Feb-2011

8.22 Framework.cpp File Reference

Class is in charge of creating and managing the master and slave processes that are going to be executed in parallel.

```
#include <exception>

#include <iostream>

#include <string>

#include <mpi.h>

#include "Algorithm.h"

#include "Defines.h"

#include "Exception.h"

#include "Framework.h"

#include "Process.h"
```

Include dependency graph for Framework.cpp:



8.22.1 Namespaces

- namespace Reconfiguration

8.22.2 Detailed Description

Class is in charge of creating and managing the master and slave processes that are going to be executed in parallel.

Author

Carmen B. Navarrete

Date

05-Jun-2009

08-Feb-2011

8.23 Framework.h File Reference

8.23.1 Classes

- class Reconfiguration::Framework

8.23.2 Namespaces

- namespace Reconfiguration

8.23.3 Detailed Description

Header file for Framework.cpp file.

Author

Carmen B. Navarrete

Date

05-Jun-2009

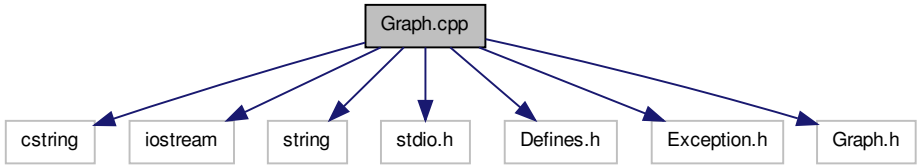
08-Feb-2011

8.24 Graph.cpp File Reference

API for managing the Graph class.

```
#include <cstring>

#include <iostream>

#include <string>

#include <stdio.h>

#include "Defines.h"

#include "Exception.h"

#include "Graph.h"
```

Include dependency graph for Graph.cpp:



8.24.1 Namespaces

- namespace Reconfiguration

8.24.2 Detailed Description

API for managing the Graph class.

Author

Carmen B. Navarrete

Date

22-abr-2009

09-Feb-2011

8.25 Graph.h File Reference

```
#include <list>
```

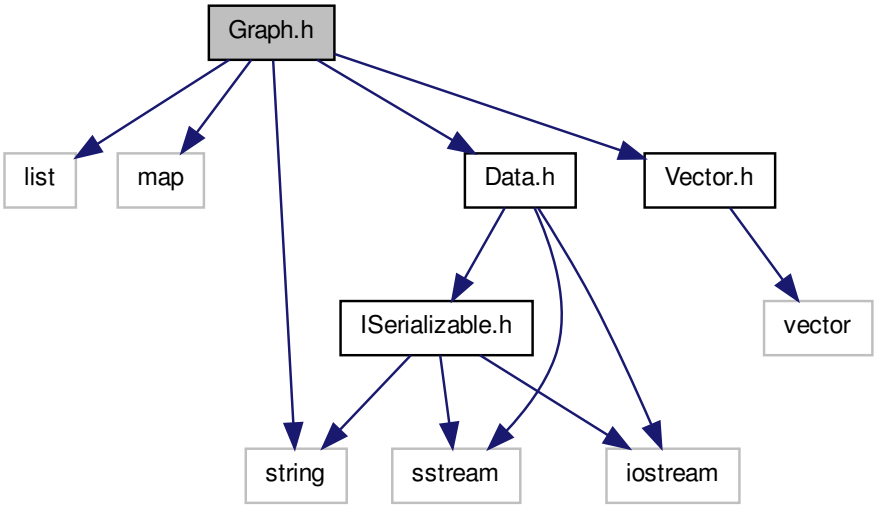
```
#include <map>
```

```
#include <string>
```

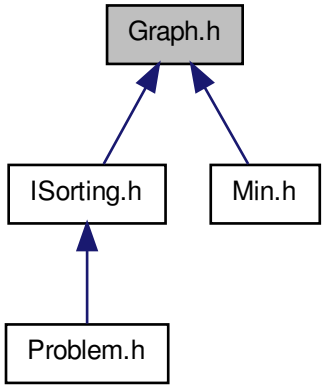
```
#include "Data.h"
```

```
#include "Vector.h"
```

Include dependency graph for Graph.h:



This graph shows which files directly or indirectly include this file:



8.25.1 Classes

- class Reconfiguration::Machine
- class Reconfiguration::Link
- class Reconfiguration::Vertex
- class Reconfiguration::Graph

8.25.2 Namespaces

- namespace Reconfiguration

8.25.3 Detailed Description

Header file for Graph.cpp file.

Author

Carmen B. Navarrete

Date

22-Abr-2009

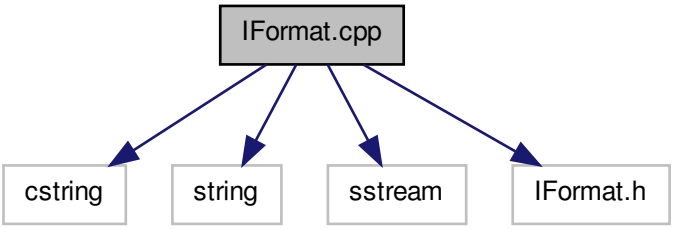
09-Feb-2011

8.26 IFormat.cpp File Reference

Implements the abstract class needed for the plugins (.so) in charge of saving the results in an output file.

```
#include <cstring>

#include <string>

#include <sstream>

#include "IFormat.h"
```

Include dependency graph for IFormat.cpp:



8.26.1 Namespaces

- namespace Reconfiguration

8.26.2 Detailed Description

Implements the abstract class needed for the plugins (.so) in charge of saving the results in an output file.

Author

Carmen B. Navarrete

Date

08-Nov-2010

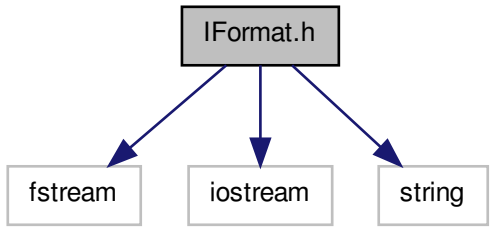
10-Feb-2011

8.27 IFormat.h File Reference

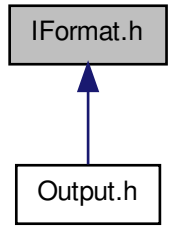
Header file for IFormat.cpp file.

```
#include <fstream>

#include <iostream>

#include <string>
```

Include dependency graph for IFormat.h:



This graph shows which files directly or indirectly include this file:



8.27.1 Classes

- class Reconfiguration::IFormat

8.27.2 Namespaces

- namespace Reconfiguration

8.27.3 Typedefs

- typedef IFormat ∗ Reconfiguration::formatcreate_t ()
- typedef void Reconfiguration::formatdestroy_t (IFormat ∗)

8.27.4 Detailed Description

Header file for IFormat.cpp file.

Author

Carmen B. Navarrete

Date

08-Nov-2010

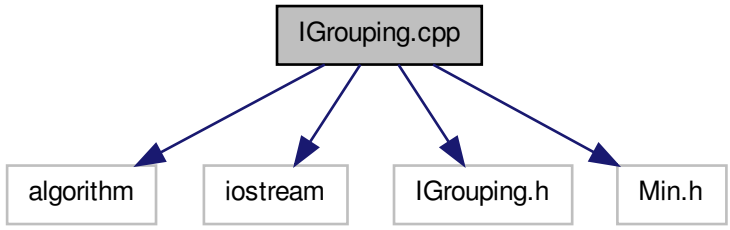
10-Feb-2011

8.28 IGrouping.cpp File Reference

API for managing the IGrouping class.

```
#include <algorithm>

#include <iostream>

#include "IGrouping.h"

#include "Min.h"
```

Include dependency graph for IGrouping.cpp:



8.28.1 Namespaces

- namespace Reconfiguration

8.28.2 Detailed Description

API for managing the IGrouping class.

Author

Carmen B. Navarrete

Date

10-May-2010

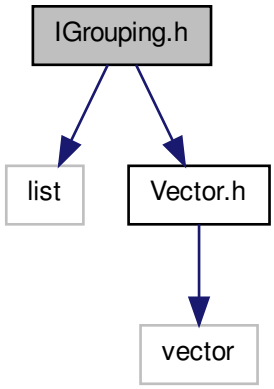
11-Feb-2011

8.29 IGrouping.h File Reference

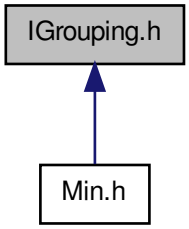
Header file for IGrouping.cpp file.

```
#include <list>

#include "Vector.h"
```

Include dependency graph for IGrouping.h:



This graph shows which files directly or indirectly include this file:



8.29.1 Classes

- class Reconfiguration::Item
- class Reconfiguration::Queue
- class Reconfiguration::IGrouping

8.29.2 Namespaces

- namespace Reconfiguration

8.29.3 Typedefs

- typedef IGrouping ∗ Reconfiguration::groupincreate_t ()
- typedef void Reconfiguration::groupingdestroy_t (IGrouping ∗)

8.29.4 Detailed Description

Header file for IGrouping.cpp file.

Author

Carmen B. Navarrete

Date

10-May-2010

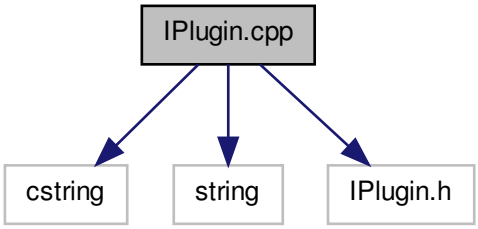
11-Feb-2011

8.30 IPlugin.cpp File Reference

Implements the abstract class needed for the plugins (.so) in charge of calculating the load-balancing.

```
#include <cstring>

#include <string>

#include "IPlugin.h"
```

Include dependency graph for IPlugin.cpp:



8.30.1 Namespaces

- namespace Reconfiguration

8.30.2 Detailed Description

Implements the abstract class needed for the plugins (.so) in charge of calculating the load-balancing.

Author

Carmen B. Navarrete

Date

09-Feb-2010

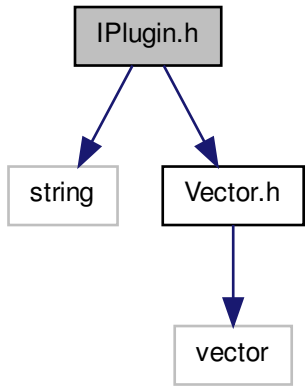
11-Feb-2011

8.31 IPlugin.h File Reference

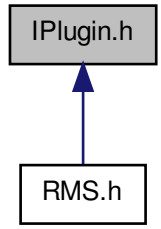
Header file for IPlugin.cpp file.

```
#include <string>

#include "Vector.h"
```

Include dependency graph for IPlugin.h:



This graph shows which files directly or indirectly include this file:



8.31.1 Classes

- class Reconfiguration::IPlugin

8.31.2 Namespaces

- namespace Reconfiguration

8.31.3 Typedefs

- typedef IPlugin * Reconfiguration::plugincreate_t ()

- typedef void Reconfiguration::plugindestroy_t (IPlugin *)

8.31.4 Detailed Description

Header file for IPlugin.cpp file.

Author

Carmen B. Navarrete

Date

09-Feb-2010

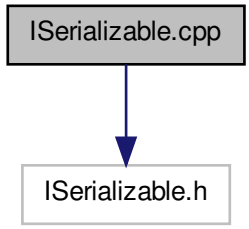
11-Feb-2011

8.32 ISerializable.cpp File Reference

Implements the abstract class needed for serialize and deserialize the user Data object.

```
#include "ISerializable.h"
```

Include dependency graph for ISerializable.cpp:



8.32.1 Namespaces

- namespace Reconfiguration

8.32.2 Detailed Description

Implements the abstract class needed for serialize and deserialize the user Data object.

Author

Carmen B. Navarrete

Date

10-May-2010

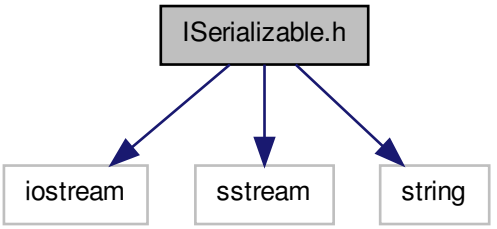
11-Feb-2011

8.33 ISerializable.h File Reference

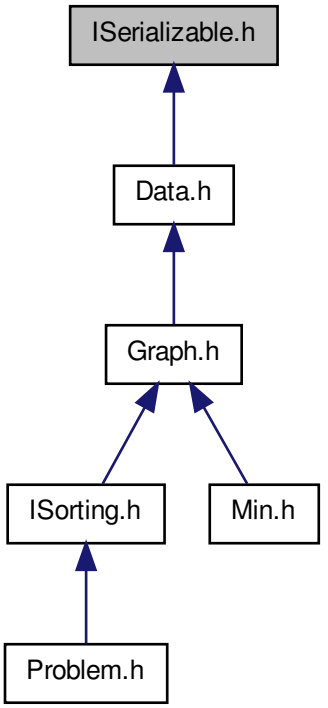
Header file for ISerializable.cpp file.

```
#include <iostream>

#include <sstream>

#include <string>
```

Include dependency graph for ISerializable.h:



This graph shows which files directly or indirectly include this file:



8.33.1 Classes

- class Reconfiguration::ISerializable

8.33.2 Namespaces

- namespace Reconfiguration

8.33.3 Detailed Description

Header file for ISerializable.cpp file.

Author

Carmen B. Navarrete

Date

10-May-2010

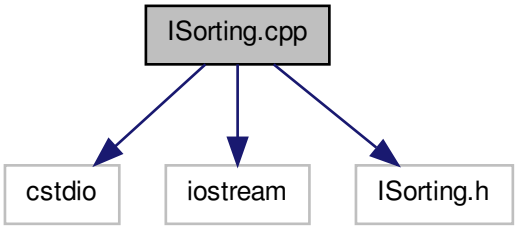
11-Feb-2011

8.34 ISorting.cpp File Reference

Implements the abstract class needed for the plugins (.so) in charge of calculating the graph decomposition.

```
#include <cstdio>

#include <iostream>

#include "ISorting.h"
```

Include dependency graph for ISorting.cpp:



8.34.1 Namespaces

- namespace Reconfiguration

8.34.2 Detailed Description

Implements the abstract class needed for the plugins (.so) in charge of calculating the graph decomposition.

Author

Carmen B. Navarrete

Date

05-Abr-2010

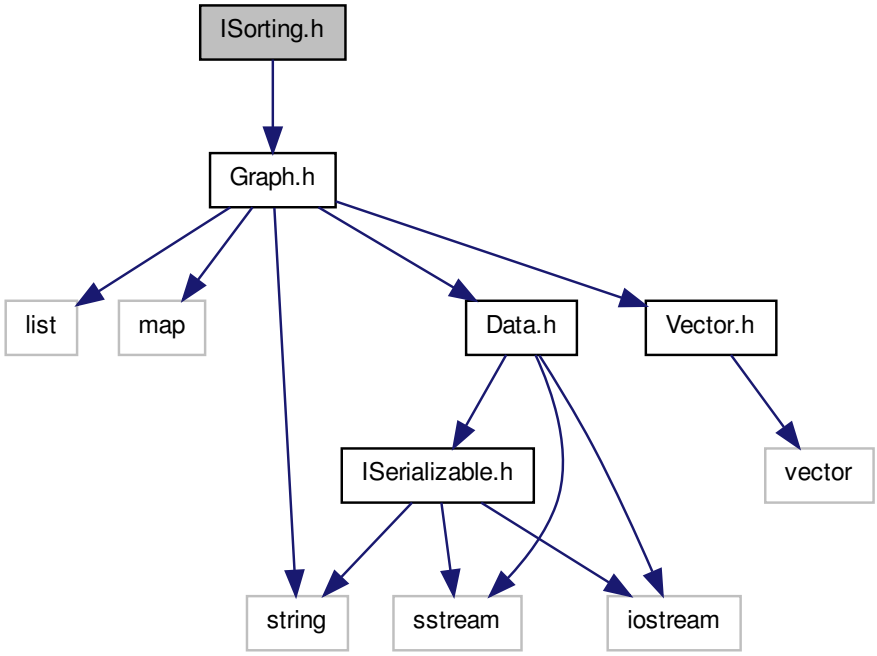
11-Feb-2011

8.35 ISorting.h File Reference

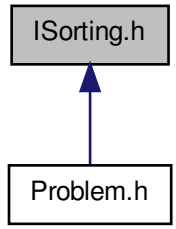
Header file for ISorting.cpp file.

```
#include "Graph.h"
```

Include dependency graph for ISorting.h:



This graph shows which files directly or indirectly include this file:



8.35.1 Classes

- class Reconfiguration::ISorting

8.35.2 Namespaces

- namespace Reconfiguration

8.35.3 Typedefs

- typedef ISorting * Reconfiguration::sortcreate_t ()
- typedef void Reconfiguration::sortdestroy_t (ISorting *)

8.35.4 Detailed Description

Header file for ISorting.cpp file.

Author

Carmen B. Navarrete

Date

05-Abr-2010

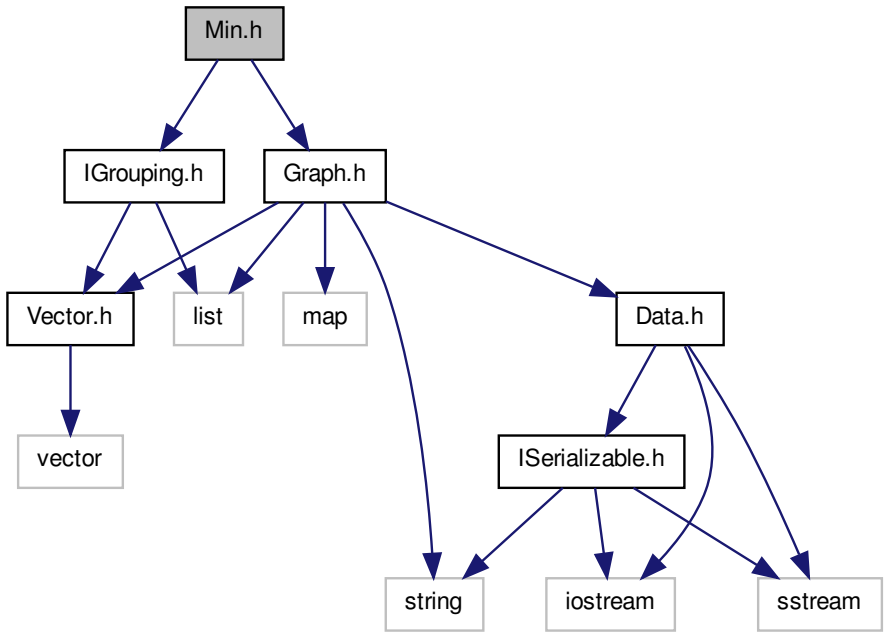
11-Feb-2011

8.36 Min.h File Reference

Miscellaneous classes for the platform.

```
#include "Graph.h"

#include "IGrouping.h"
```

Include dependency graph for Min.h:



8.36.1 Classes

- class Reconfiguration::Min

8.36.2 Namespaces

- namespace Reconfiguration

8.36.3 Detailed Description

Miscellaneous classes for the platform.

Author

Carmen B. Navarrete

Date

30-Jul-2009

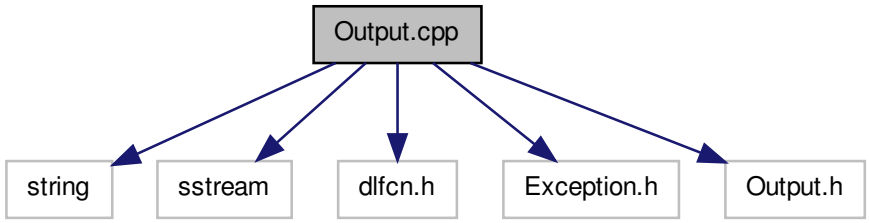
21-Feb-2011

8.37 Output.cpp File Reference

Manages the output plugins called by the user and its handlers.

```
#include <string>

#include <sstream>

#include <dlfcn.h>

#include "Exception.h"

#include "Output.h"
```

Include dependency graph for Output.cpp:



8.37.1 Namespaces

- namespace Reconfiguration

8.37.2 Detailed Description

Manages the output plugins called by the user and its handlers.

Author

Carmen B. Navarrete

Date

13-Jul-2009

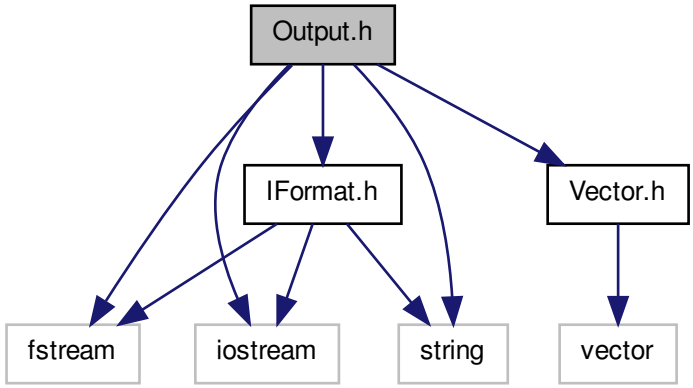
13-Feb-2011

8.38 Output.h File Reference

Header file for Output.cpp file.

```
#include <fstream>

#include <iostream>

#include <string>

#include "IFormat.h"

#include "Vector.h"
```

Include dependency graph for Output.h:



8.38.1 Classes

- class Reconfiguration::Output

8.38.2 Namespaces

- namespace Reconfiguration

8.38.3 Detailed Description

Header file for Output.cpp file.

Author

Carmen B. Navarrete

Date

13-Jul-2009

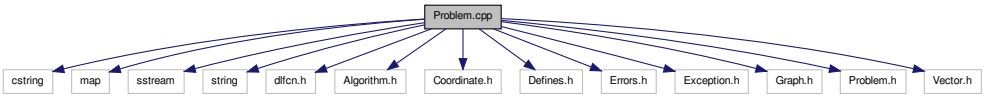
13-Feb-2011

8.39 Problem.cpp File Reference

API for managing the Problem class.

```
#include <cstring>

#include <map>

#include <sstream>

#include <string>

#include <dlfcn.h>

#include "Algorithm.h"

#include "Coordinate.h"

#include "Defines.h"

#include "Errors.h"

#include "Exception.h"

#include "Graph.h"

#include "Problem.h"

#include "Vector.h"
```

Include dependency graph for Problem.cpp:



8.39.1 Namespaces

- namespace Reconfiguration

8.39.2 Detailed Description

API for managing the Problem class.

Author

Carmen B. Navarrete

Date

31-Mar-2009

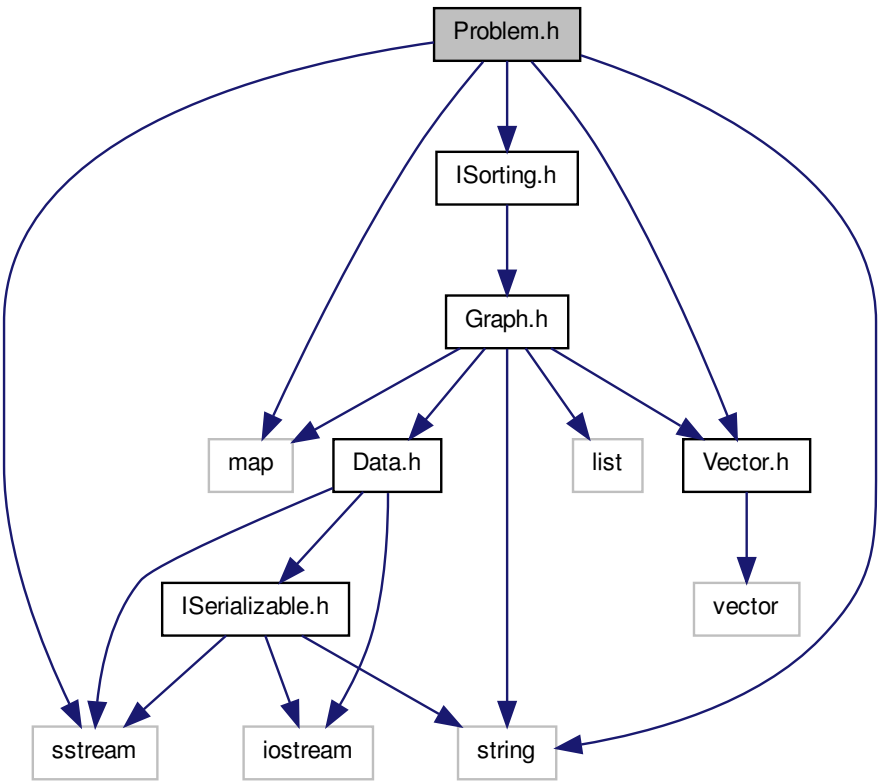
15-Feb-2011

8.40 Problem.h File Reference

Header file for Problem.cpp file.

```
#include <map>

#include <sstream>

#include <string>

#include "ISorting.h"

#include "Vector.h"
```

Include dependency graph for Problem.h:



8.40.1 Classes

- class Reconfiguration::MapElements
- class Reconfiguration::Partition

- class Reconfiguration::Problem

8.40.2 Namespaces

- namespace Reconfiguration

8.40.3 Detailed Description

Header file for Problem.cpp file.

Author

Carmen B. Navarrete

Date

31-Mar-2009

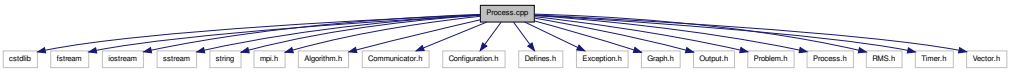
15-Feb-2011

8.41 Process.cpp File Reference

Manages the MPI processes and its execution.

```
#include <cstdlib>

#include <fstream>

#include <iostream>

#include <sstream>

#include <string>

#include <mpi.h>

#include "Algorithm.h"

#include "Communicator.h"

#include "Configuration.h"

#include "Defines.h"

#include "Exception.h"

#include "Graph.h"

#include "Output.h"

#include "Problem.h"

#include "Process.h"

#include "RMS.h"

#include "Timer.h"

#include "Vector.h"
```

Include dependency graph for Process.cpp:



8.41.1 Namespaces

- namespace Reconfiguration

8.41.2 Variables

- int flag = 0

8.41.3 Detailed Description

Manages the MPI processes and its execution.

Author

Carmen B. Navarrete

Date

11-Ago-2009

17-Feb-2011

8.41.4 Variable Documentation

int flag = 0

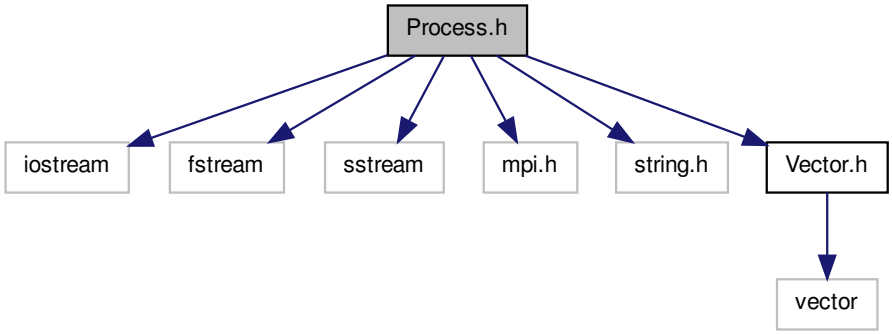
Global flag to determine when the slave ends.

8.42 Process.h File Reference

Header file for Process.cpp file.

```
#include <iostream>

#include <fstream>

#include <sstream>

#include <mpi.h>

#include <string.h>

#include "Vector.h"
```

Include dependency graph for Process.h:



8.42.1 Classes

- class Reconfiguration::Process
- class Reconfiguration::SuperMaster
- class Reconfiguration::Slave

8.42.2 Namespaces

- namespace Reconfiguration

8.42.3 Detailed Description

Header file for Process.cpp file.

Author

Carmen B. Navarrete

Date

11-Ago-2009

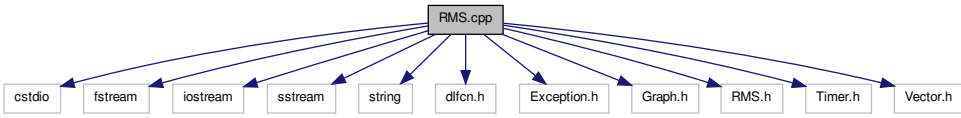
17-Feb-2011

8.43 RMS.cpp File Reference

Manages the loadbalancing plugins.

```
#include <cstdio>

#include <fstream>

#include <iostream>

#include <sstream>

#include <string>

#include <dlfcn.h>

#include "Exception.h"

#include "Graph.h"

#include "RMS.h"

#include "Timer.h"

#include "Vector.h"
```

Include dependency graph for RMS.cpp:



8.43.1 Namespaces

- namespace Reconfiguration

8.43.2 Detailed Description

Manages the loadbalancing plugins.

Author

Carmen B. Navarrete

Date

13-May-2009

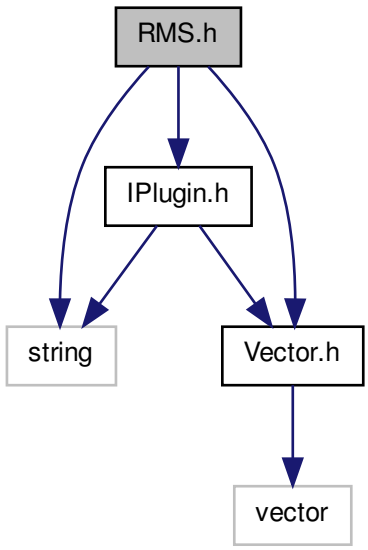
21-Feb-2011

8.44 RMS.h File Reference

Header file for RMS.cpp file.

```
#include <string>

#include "IPlugin.h"

#include "Vector.h"
```

Include dependency graph for RMS.h:



8.44.1 Classes

- class Reconfiguration::RMS

8.44.2 Namespaces

- namespace Reconfiguration

8.44.3 Detailed Description

Header file for RMS.cpp file.

Author

Carmen B. Navarrete

Date

13-May-2009

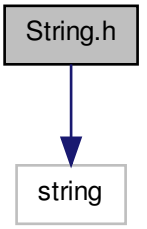
21-Feb-2011

8.45 String.h File Reference

Extends the features of the standard string class.

```
#include <string>
```

Include dependency graph for String.h:



8.45.1 Classes

- class Reconfiguration::String

8.45.2 Namespaces

- namespace Reconfiguration

8.45.3 Detailed Description

Extends the features of the standard string class.

Author

Carmen B. Navarrete

Date

24-Jun-2009

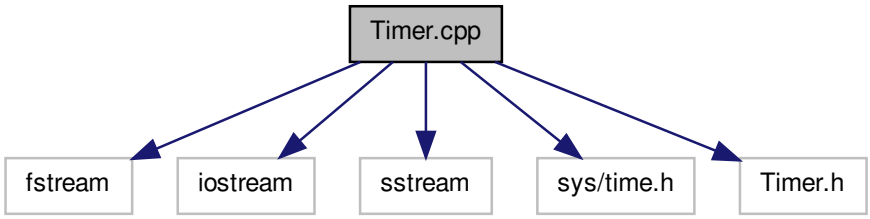
21-Feb-2011

8.46 Timer.cpp File Reference

Platform timer clock.

```
#include <fstream>

#include <iostream>

#include <sstream>

#include <sys/time.h>

#include "Timer.h"
```

Include dependency graph for Timer.cpp:



8.46.1 Namespaces

- namespace Reconfiguration

8.46.2 Detailed Description

Platform timer clock.

Author

Carmen B. Navarrete

Date

13-May-2009

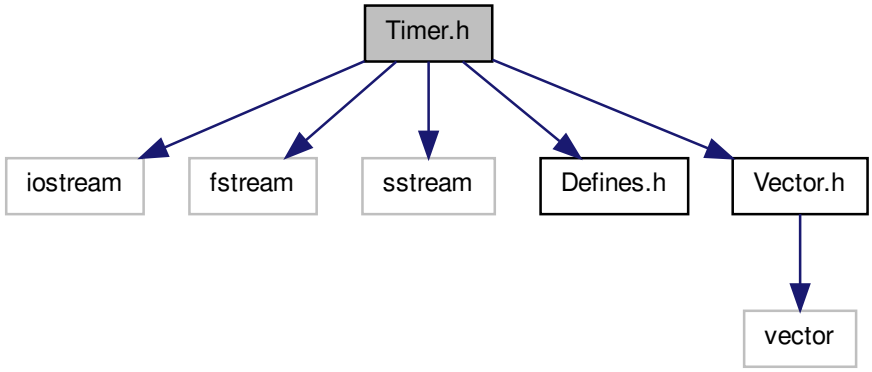
21-Feb-2011

8.47 Timer.h File Reference

Header file for Timer.cpp file.

```
#include <iostream>

#include <fstream>

#include <sstream>

#include "Defines.h"

#include "Vector.h"
```

Include dependency graph for Timer.h:



8.47.1 Classes

- class Reconfiguration::Timer

8.47.2 Namespaces

- namespace Reconfiguration

8.47.3 Detailed Description

Header file for Timer.cpp file.

Author

Carmen B. Navarrete

Date

13-May-2009

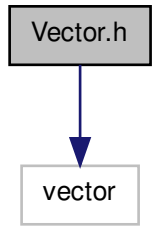
21-Feb-2011

8.48 Vector.h File Reference

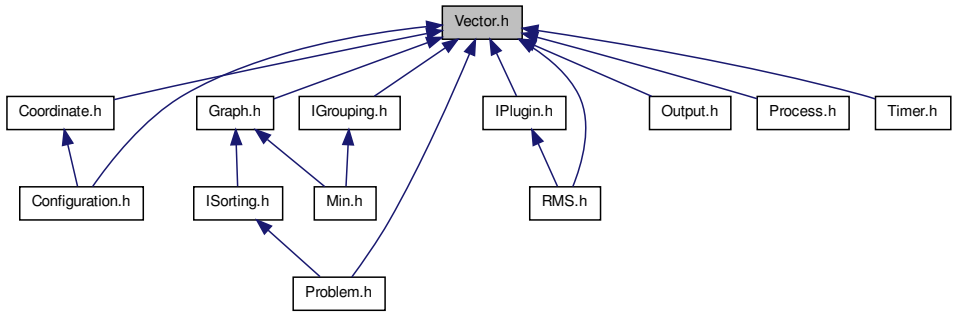
Extends the features of the standard vector class.

```
#include <vector>
```

Include dependency graph for Vector.h:



This graph shows which files directly or indirectly include this file:



8.48.1 Classes

- class Reconfiguration::Vector< object >

8.48.2 Namespaces

- namespace Reconfiguration

8.48.3 Detailed Description

Extends the features of the standard vector class.

Author

Carmen B. Navarrete

Date

09-Jun-2009

21-Feb-2011